

Université - Clermont-Ferrand

École Doctorale des Sciences Pour l'Ingénieur

CEA - LIST

LCSR

Adaptation du comportement sensori-moteur de robots mobiles en milieux complexes

HILL Ashley

Soutenue publiquement le 24 Novembre 2022

Membres du jury:

Anne AUGER, Présidente, Ecole Polytechnique
David FILLIAT, Rapporteur, ENSTA Paris
Faïz BEN AMAR, Rapporteur, Sorbonne Université
Eric LUCET, Examineur, CEA Saclay
Roland LENAIN, Directeur de thèse, INRAE Clermont-Ferrand



*To my parents,
To my Fiancée Alizée*

Abstract

This dissertation addresses the fullest possible adaptability of mobile robots following a path in an off-road context. Indeed, this thesis was born from the need to continuously adapt the behavior of a vehicle, according to variations in the quality of sensor perception and grip conditions. Classically, this is achieved by using increasingly complex sensor and control systems. However, this approach optimizes the sensor and control aspects independently, whereas in reality a strong correlation exists between them, resulting in sub-optimal control strategies.

The modeling of the system and the design of the existing control laws and observers are first described, which contribute to the closed-loop control of the robot. From this, a baseline of an unmodified control system can be tested and validated both in real world and in simulation. In addition, this description highlights one deterministic approach to improving the online adaptability of mobile robot path tracking.

A machine learning approach is then considered. This is a reinforcement learning approach with episodic policy iterations using an evolutionary strategy, that is used to train a neural network. The machine learning training is then exploited to improve the existing control laws by taking into account additional inputs such as sensor accuracy and grip conditions, whose effective contribution is evaluated.

Different methods of using the neural network are considered. A complete replacement of the steering control law is proposed. An alternative approach of online adjustment of the steering control parameters allows the original robust control law to be preserved while using additional information.

These methods are tested in simulation and in real-life conditions, as well as a deterministic model based control parameter tuning approach. This analysis reveals the specific strengths and weaknesses of each approach, with respect to the baseline methods. Further analysis are also performed using a feature importance method developed during the thesis, which allow some insights into the behavior of the neural network.

It is observed that an augmentation of the steering control system alone is sub-optimal. A second approach of using a neural network for both steering and speed control is then designed and compared. For this approach, more care is needed in order to develop the appropriate objective function to achieve suitable trade-offs, due to the characteristics of the Pareto front for this multi-objective optimization approach.

Overall, the machine learning hybrid control approaches developed in this thesis have been tested through real world experiments in highly dynamic off-road environments, with varying grip conditions and sensor accuracy. The resulting findings show that these methods are able to outperform existing controllers in both highly varying and constant environments, demonstrating that the proposed method is capable of adapting the robot's behavior in a strong manner, relative to its observed state.

Keywords: Wheeled mobile robots, off-road robotics, path tracking, adaptive control, non-linear control, machine learning, reinforcement learning, neural networks, evolutionary strategies, gain tuning, dynamic simulator, optimization, Pareto front, feature importance.

Résumé

Cette thèse s'intéresse à l'adaptabilité la plus complète possible des robots mobiles suivant une trajectoire dans un contexte hors route. En effet, cette thèse est née de la nécessité d'adapter continuellement le comportement d'un véhicule, en fonction des variations de la qualité de perception des capteurs et des conditions d'adhérence. Classiquement, ceci est réalisé en utilisant des systèmes de capteurs et de contrôle de plus en plus complexes. Cependant, cette approche optimise les aspects capteurs et contrôle de manière indépendante, alors qu'en réalité une forte corrélation existe entre eux, résultant en des stratégies de contrôle sous-optimales.

La modélisation du système et la conception des lois de commande et des observateurs existants sont d'abord décrites, ce qui contribue à la commande en boucle fermée du robot. À partir de là, une base de référence d'un système de contrôle non modifié peut être testée et validée à la fois dans le monde réel et en simulation. En outre, cette description met en évidence une approche déterministe pour améliorer l'adaptabilité en ligne du suivi de trajectoire des robots mobiles.

Une approche par apprentissage est ensuite envisagée. Il s'agit d'une approche d'apprentissage par renforcement avec des itérations épisodiques de la politique à l'aide d'une stratégie évolutionnaire, qui est utilisée pour former un réseau de neurones. L'apprentissage est ensuite exploité pour améliorer les lois de commande existantes en prenant en compte des données supplémentaires telles que la précision des capteurs et les conditions d'adhérence, dont la contribution effective est évaluée.

Différentes méthodes d'utilisation du réseau de neurones sont envisagées. Un remplacement complet de la loi de contrôle de l'angle de braquage est proposé. Une approche alternative d'ajustement en ligne des paramètres de contrôle de la direction permet de préserver la loi de contrôle robuste tout en utilisant les informations supplémentaires.

Ces méthodes et une méthode de réglage des paramètres de contrôle basée sur un modèle déterministe sont testées en simulation et en conditions réelles. Cette analyse révèle les forces et faiblesses spécifiques de chaque approche, par rapport aux méthodes de base. D'autres analyses sont également effectuées en utilisant une méthode d'importance des entrées du réseau de neurones développée au cours de la thèse, ce qui permet de mieux comprendre le comportement du réseau de neurones.

Il est constaté qu'une amélioration du système de contrôle de la direction seule est sous-optimale. Une deuxième approche consistant à utiliser un réseau de neurones pour le contrôle de la direction et de la vitesse simultanément est alors conçue et comparée. Pour cette approche, une plus grande attention est nécessaire afin de concevoir la fonction objectif appropriée pour obtenir les compromis adéquats, en raison des caractéristiques du front de Pareto pour cette approche d'optimisation multi-objectifs.

Dans l'ensemble, les approches de contrôle hybride par apprentissage développées dans cette thèse ont été testées par des expériences réelles dans des environnements hors route hautement dynamiques, avec des conditions d'adhérence et une précision des capteurs variables. Les résultats montrent que ces méthodes sont capables de surpasser les contrôleurs existants dans des environnements à la fois très variables et constants, démontrant que la méthode proposée est capable d'adapter le comportement du robot de manière importante, par rapport à son état observé.

Motclés: Robots mobiles à roues, robotique tout-terrain, suivi de trajectoire, commande adaptative, commande non linéaire, apprentissage, apprentissage par renforcement, réseaux de neurones, stratégies évolutionnaires, réglage de gain, simulateur dynamique, optimisation, front de Pareto, importance des entrées.

Thanks

This thesis is the results of the cumulative works over 3 years at the French Alternative Energies and Atomic Energy Commission (CEA) in Saclay, with many back and fourths to the National Research Institute for Agriculture Food and Environment (INRAE) in Clermont-Ferrand. As such, I would like to thank these institutes for their warm welcome, and for the means they gave me to lead to work to fruition.

My sincerest thanks to Anne Auger, for accepting to be the president of the jury for my thesis, along with David Filliat and Faïz Ben Amar for being my rapporteur de thèse. Their guidance and advice over the PhD was invaluable while working on this subject.

I wish to wholeheartedly thank Roland Lenain, who was my PhD director. He showed inspiring passion for science, rigor, was very patient while stopping me running headfirst, and was very kind. Most importantly he knew how to formalize my ideas, inspirations, and showed me how to get to when I needed to go. I look forward working with him again, watching my Neural networks running and then threatening to throw themselves into a ditch, along with a nice quack for good measure.

I would like show my deepest gratitudes to Eric Lucet, my PhD advisor. He saw me at my best, encouraging me forward, and he saw me at my worst, knowing exactly what to say to get me going again. His Foresight and intuition are awe-inspiring. I hope to keep brainstorming and to bounce ideas off him, to push scientific subjects further.

To the Dr Hamelin, Dr Bretel, Dr Martins, and Dr Martin, I thank them for giving me their time and advice on how to finish a PhD while remaining (somewhat) sane, and I thank them for showing me that I was not alone in my struggles. And I hope the next time I meet them, it can count as a symposium. Drinks on me.

To everyone who read parts of my thesis (you know who you are), I thank you for the invaluable time and advice that allowed the thesis to be as clean as it is here. Without them this document would not be in a readable and coherent state.

To my fiancée Alizée, I cannot express the gratitude I owe her. she saw me from the start to the end, and showed me when I was metaphorically (and sometimes literally) banging my head against a wall, that I had not reconsidered my initial hypothesis, leading me to bring many of my ideas to fruition.

I would like to sincerely thank my parents, my sisters, and my family, for helping me become the person I am today, teaching me things I will never forget, for encouraging me to do what I love, and to have helped me to finish this work.

I would also like to thank Jeff (The Guardsman), Pierre (le papillon), Dimity (l'oracle), Jordan (Le Firner), Pierre-Louis (Le Fisc), and my other friends for their good spirits and their help to blow off some steam around some games, hopefully I won't lose too badly next time.

I wish to thank the Factory-IA cluster, for simulating 800'000km and 6.5 years per experiment over 24 hours, totaling over 800'000'000km (5.3 Au) and 650 years, which is the equivalent of going from the sun to Jupiter over the same time-span as from the middle ages to today. Without it, I do believe my laptop would have caught fire... And I would like to thank Robufast & Adap2e for their discipline, consistency, and for 30GB of data over 18 days of experiments. I promise (or I will try to) not throw them in a ditch by accident because I loaded the wrong tyre model for the neural network (don't ask).

Finally, I want to thank specifically John Tatman, Vanessa Tatman, my fiancée Alizée, My sisters Claire & Louise, Jeffrey Hannan, Thibault Hamelin, Tolentino Martins, and Lucie Martins during some dark times near the end of the thesis. Thank you for keeping the lights on.

I do not know what I may appear to the world, but to myself I seem to have been only like a boy playing on the seashore, and diverting myself in now and then finding a smoother pebble or a prettier shell than ordinary, whilst the great ocean of truth lay all undiscovered before me.

Sir Isaac Newton

[They] offered to give us our future. [We] will achieve [our] own future. Technology is not a straight line. There are many paths to the same end. Accepting another's path blinds you to alternatives.

Legion, ME2

Table 0.1: Notation used throughout the dissertation

<u>Robotics:</u>	
(D)	The trajectory followed by the robot.
s	The curvilinear abscissa along (D) .
L	The wheel base length of the robot.
v	The amplitude of the speed vector of the robot.
P_x, P_y	The x, y position in a global reference frame.
θ	The robot's heading.
$\tilde{\theta}$	The angular error.
y	The lateral error.
$c(s)$	The curvature at the curvilinear abscissa s .
δ_F	The state of the front steering angle.
δ_R	The state of the rear steering angle.
g	The gravitational constant (defined here as $g = 9.81$).
\mathbf{a}	The longitudinal robot acceleration amplitude.
G	The center of gravity of the robot.
L_F, L_R	The front and rear wheelbase of the robot respectively.
F_F, F_R	The front and rear lateral wheel force respectively.
β	The robot's sliding angle.
β_F, β_R	The robot's front and rear sliding angles respectively.
I_z	The moment of inertia across the Z axis.
v_2	The longitudinal component of the speed vector of the robot.
$\ddot{\theta}$	The angular acceleration across the Z axis of the robot.
$\dot{\theta}$ & ω	The angular speed across the Z axis of the robot.
m	The mass of the robot.
C_F, C_R	The cornering stiffness of the front and rear wheels of the robot respectively.
u_{δ_F} & $\delta_{ctrl,F}$	The control input of the front steering angle.
τ_s	The time constant to convergence.
K_p	The proportional control gain.
K_d	The derivative control gain.
H	The control horizon lookahead (in seconds).
Δt	The time between two measurements.
Δs	The distance along the curvilinear abscissa between two measurements.
dt	The differential change over time.
\hat{x}	The predicted state vector.
$f()$	The system's model.
x	The state vector.
u	The control vector.
F_k	The Jacobian matrix of the robot's model.
\mathcal{C}	The Kalman filter's covariance matrix.
\hat{x}'	The estimated state vector.
K'	The Kalman gain.
H_k	The Jacobian matrix of the observation model.
z_k	The measurements used for the Kalman filter.
Q	The covariance matrix of the robot's model.
R	The covariance matrix of the measurements.
K_{pos}	The sliding angle observer gain over the position.
K_β	The sliding angle observer gain over the sliding angle.
A	The matrix of the system's state model.
B	The matrix of the system's control model.
k_{dd}	The double derivative gain (used for EBSF).
ξ	The damping factor of the control system.
τ_δ	The steering actuator response time.
D_y	The settling distance for the convergence of the lateral error.
T_y	The settling time for the convergence of the lateral error.
T_ω	The settling time for the convergence of the angular velocity.
π	The ratio between the diameter and the perimeter of a circle.
N	The constant between two values for Shannon's sampling theory.

<u>Machine learning:</u>	
$\pi()$	The policy function.
X	The state vector.
\mathbf{s}	The observed state.
a	The action vector.
r	The reward.
$V()$	The value function.
$Q()$	The Q-value function.
T	The total time of an episode.
\mathcal{P}	The population sampled for an optimizer.
\mathcal{N}	The normal distribution function.
μ	The mean of a Gaussian distribution.
σ	The standard deviation of a Gaussian distribution.
C	The covariance matrix of CMA-ES.
I	The identity matrix.
N_{pop}	The population size of CMA-ES.
G_t	The optimization target for reinforcement learning.
<u>Overall:</u>	
N	The total number of samples in an episode (where $T = \sum_{k=0}^N dt$).
s_N	The total length of a trajectory.
obj_{err}	The sub-objective function that describes the lateral error.
obj_{steer}	The sub-objective function that describes the steering error.
obj_{speed}	The sub-objective function that describes the speed penalty.
k_y	The objective function lateral error gain.
k_{steer}	The objective function steering error gain.
k_{speed}	The objective function speed gain.
γ	The objective function linear scalarization coefficient.
obj_1	The first objective function used in section 4 and section 5.
$obj_{1,speed}$	The objective function used in section 6.1.
obj_2	The second objective function used in section 6.1.
obj_3	The third objective function used in section 6.
A_{error}	The surface error described in m^2 .
A_{over}	The surface error outside the allowed corridor, described in m^2 .
\bar{v}	The average speed over the episode.
\mathcal{C}_{xy}	The x, y position accuracy, as described by the Kalman covariance matrix.
<u>Pacejka sliding model: (only in section 2.3)</u>	
F_y	The lateral wheel force.
F_z	The vertical force applied to the wheels.
α	The sliding angle (in degrees).
B, C, D, E	The Pacejka parameters.
a_1, \dots, a_8	The Pacejka parameters.
<u>Romea controller: (only in section 2.5)</u>	
a_1, a_2, a_3	The state variables, representing the curvilinear abscissa, the lateral error, and the angular error respectively.
m_1, m_2, m_3	The control variables, representing the speed, front steering, and a new control variable relying on the steering angle respectively.
<u>EBSF controller: (only in section 2.5)</u>	
\mathbf{U}	The control input vector.
\mathbf{y}_0	The immediate error state vector.
$\mathcal{A} \ \& \ \mathcal{B}$	The matrices of the system linearized predicted states ($\mathbf{Y} = \mathcal{A}\mathbf{y}_0 + \mathcal{B}\mathbf{U}$).
\mathbf{Y}	The predicted error state vectors.
\mathfrak{D}	The block diagonal matrix that contains the dimensions of the robot represented as a rectangle.
\mathfrak{d}_{gap}	The vector of the gap size that the constraint must respect.
$\mathcal{Q} \ \& \ \mathcal{R}$	The block diagonal gain matrices with elements $\gamma_Q^k Q$ and $\gamma_R^k R$ respectively.
S	The discretization step over the curvilinear abscissa.
n	The number of steps chosen to define the prediction horizon.

Contents

Contents	11
1 Introduction	15
1.1 Towards intelligent systems	15
Is it possible to define intelligence?	15
Neurons and uses for intelligent systems	16
Artificial Intelligence and Robotics, from fiction to reality	16
Some historical perspectives regarding Artificial intelligence	17
Limitations of Control theory and AI applied to robotic control	18
1.2 Methods for adapting the control in complex environments using machine learning	19
The context of the thesis	19
Research axis	19
Implications of machine learning	20
Applying deep reinforcement learning to mobile robotics	21
2 Vehicle Modeling and control	23
2.1 General features about modeling	23
2.2 Kinematic Model	23
2.3 Dynamic model	24
Improving the Kinematic model	24
Tyre slip model	26
Actuators delays	27
2.4 Extended kinematic model	27
2.5 Deterministic steering control	28
Adaptive control law from a chained system [Romea]	28
predictive control law from constraint optimization [EBSF]	29
Tuning control laws parameters	30
2.6 Extended Kalman filter	31
2.7 Observers	32
Sliding angles observer	32
Cornering stiffness observer	33
2.8 Simulated implementation of the models and controllers	33
3 Reinforcement learning approach to robotic control	37
3.1 Overview of the machine learning methods	37
Self-supervised learning	37
Supervised learning	37
Reinforcement learning	37
A Markov modeling for robotic control	38
3.2 Time difference reinforcement learning	39
Value function	39
Action policy	39
Existing methods	40
Limitations	41
3.3 Transition to episodic	42
3.4 Gradient-free Direct policy search	43

	An alternative to time difference	43
	Moving from reward to objective function	43
	Optimizers for episodic reinforcement learning	44
3.5	CMA-ES based training in simulation	46
3.6	Neural network architecture	46
3.7	Reinforcement learning strategy selection	47
4	Applying reinforcement learning for robotic steer control	49
4.1	Direct steer control using Reinforcement learning [<i>NN controller</i>]	49
	Experimental setup	49
	Simulated results	51
	Feature importance	54
	Analysis of the approach	55
4.2	Corrective steer control [<i>Delta NN ctrl</i>]	55
	Experimental setup	56
	Simulated results	57
	Feature importance	59
	Analysis of the approach	60
4.3	Online control parameter tuning for existing steer controller [<i>NN gain tuner</i>] . .	60
	Control parameter tuning	61
	Experimental setup	63
	Simulated results	64
	Feature importance	67
	Validation of the results over test trajectories	68
	Analysis of the approach	69
5	Gain tuning in dynamic context	71
5.1	Model-based gain tuning [<i>Model gain tuner</i>]	71
	System response time	71
	Settling time for the robots yaw rate	72
	Gain adaptation	72
	Experimental setup	73
	Metrics	73
	Simulated results	73
	Analysis of the approach	76
5.2	Control parameter tuning using dynamic parameters [<i>Full NN gain tuner</i>] . . .	77
	Experimental setup	77
	Simulated results	78
	Qualitative Analysis	79
	Feature importance	83
	Validation of the results over test trajectories	84
	Analysis of the approach	84
5.3	Real world experiments	86
	The RobuFast robotic platform	86
	Experimental Setup	86
	Trajectory 1	87
	Trajectory 2	88
	Conclusion	89
6	Simultaneous steer and speed control	91
6.1	The problem shift due to additional speed control	91
	Pareto Front	92
	New Objective function	94
6.2	Experimental setup	95
	Control loop setup	95
	Metrics	96
	Training details	97
6.3	Simulated results	97

Quantitative Analysis	97
Qualitative Analysis	99
Feature importance	101
Validation of the results over test trajectories	103
Analysis of the approach	104
6.4 Real world experiments	105
Experimental setup	105
Real world results	106
Analysis of the results	108
7 Conclusion and Future works	109
7.1 Conclusions	109
7.2 Future works & perspectives	111
Tuning the model based gain tuner	111
Improving the observations	111
Alternate architecture for integrating the neural network	111
Improving the simulation for additional dynamics	112
Predicting the settling time with a neural network, for agnostic controller gain tuning	112
Gain tuning: going further than controllers	112
Speed control applied independently to each wheel	112
Transformer applied to robotic control	113
Custom Neural network architecture	113
Improving the optimizer	113
7.3 Overview of the work	113
Bibliography	115
List of Figures	121
A Appendices	127
A.1 TD Reinforcement Learning: Function derivation	127
Optimization target: G_t	127
Value function: $V(\mathfrak{s})$	127
Q-value: $Q(\mathfrak{s}, a)$	128
A.2 Comparing optimizer algorithms for mobile robot steering	129
BSR: Basic Random search	129
CEM: Cross-Entropy Method	129
An empirical test: Comparing with CMA-ES	130
A.3 CMA-ES analysis	133
covariance exploration	133
CMA-ES variants	133
CMA-ES limitations for real world experimentation	135
A.4 New feature importance method	137
Feature importance	137
Temporal permutation	137
Novel gradient base approach	137
Deriving linear approximations	138
Deriving N-order Taylor approximations	138
Gradient base feature importance of experimental results	139
A.5 <i>NN controller</i> and <i>Delta NN ctrl</i> with dynamic parameters	141
A.6 Training method variance test: <i>Full NN gain tuner</i> case study	143
A.7 Gain synthesis using CMA-ES in dynamic simulation	145
Experimental setup	145
Results	146
Limitations	149
A.8 Real world trials with dynamic parameters	151
Trials with the Pacejka tyre model for training	151
A.9 Online speed and control parameter tuning up to 6m/s, with linear objective function	155

	Overview of the experiment	155
	Trajectory 1	155
	Trajectory 2	159
	Testing a pure machine learning controller	161
	Conclusion of supplementary experiments	162
A.10	Simulator implementation and tools	165
	Trajectory format	166
	Tools for analysis	166
A.11	Society's feelings and expectations regarding artificial intelligences and robotics . .	169
A.12	Description of elementary trajectories for training and testing	171
	Training set	171
	Testing set	171

Chapter 1

Introduction

The subject of interest considered here is the autonomous navigation of a wheeled mobile robot in off-road environment. To this end, the methodology investigated is to complete classical control theory using the latest developments in machine learning.

This work is the result of a collaboration between the CEA and INRAE research institutes. It was initiated and co-supervised by the CEA while the INRAE provided the direction for the PhD. These works took place in the two laboratories:

- The LCSR which is part of the DRT/LIST/DIASI/SRI in the CEA Saclay, specialized in research for the control of robots in industrial applications.
- The TSCF which is part of the INRAE Clermont, which is specialized in research for the control of mobile robots in an agricultural context.

The following work have been achieved in part using with Factory-IA in CEA Nano-INNOV at Palaiseau, and real world off-road tests took place in INRAE's dedicated site at Montoldre.

1.1 Towards intelligent systems

To begin with, the concepts of AIs and machine learning are introduced, before specifying the use case of said methods over the research axis that are considered.

Is it possible to define intelligence?

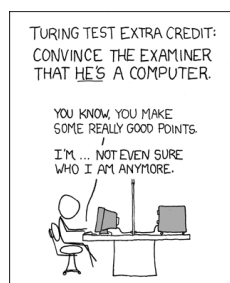


Figure 1.1: XKCD's spin on Turing tests.

One of the hardest questions that are yet to be fully answered, is "*What is intelligence?*" and the companion question "*How can one quantify intelligence?*". Indeed these two questions are large, complex, and very subjective, that have plagued many minds. Yet it can be known when observed, in a "I know it when I see it" fashion, without any clear concepts to attribute to these kinds of answers, which implies that intelligence is a concept with no hard boundaries (this feature of intelligence is what Alan Turing exploits for his famous Turing test [1], where a human is asked to guess if a system is a human or a machine).

Over the years, many attempts to quantify intelligence have been explored, including IQ (intelligence quotient) tests [2], but these tests have shown to not be a useful indicator of intellectual capabilities [3], and are often an indication of competence in a very narrow subset of intelligence. Finding an objective method for quantifying intelligence seems intractable for the fields of philosophy, neuroscience, and social sciences.

However, it is not necessary to attempt to quantify what is intelligence, and to simply define what *is* intelligence. In computer science and AI research, a common definition exists:

"An intelligent system is a system capable of maximizing a desired objective within a given environment."

However this definition is not as useful as it seems. On the surface this definition seems valid, as it takes into account human intelligence, but also animal intelligence, and advance artificial intelligences such as Alpha-Go [4]. Unfortunately, it is also quite large and includes simple organisms such as amoeba and viruses as being intelligent even though they lack a central nervous system needed for abstract reasoning and planning. As such, it might be more useful to develop an intuition of intelligence for the reader, rather than trying to define or quantify it, by taking example on Alan Turing's *Turing test*.

Neurons and uses for intelligent systems

In nature, there are many examples of intelligent systems. Indeed, if we consider humans to be intelligent, an example of an intelligent system can be the human central nervous system. A nervous system is composed of neurons with axons (used to connect neurons between each other) as shown in figure 1.2. Individually, these neurons are not inherently intelligent. Indeed, their behavior is rather simple: they activate their axon if the axons are connected to the neurons, signal above the threshold of the neuron. A few neurons connected can allow for simple sense-action loops, which are the basis for some microscopic creatures (for example if sensing enough food on the left, then move towards the left). Neurons take a whole new form when combining in large networks, due to the emergent behavior of the neurons together allowing for a higher level of intelligence, often associated with central nervous systems seen in larger animals. Indeed, this emergent intelligence is capable of long and short term planning, problem solving, and abstract reasoning, which are features that are not present within the basic neuron building block.

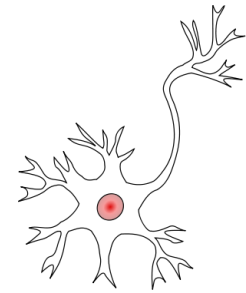


Figure 1.2: A diagram of a neuron.

These qualities are often necessary in computer sciences and in particular in robotics, as they would allow for complex automation without the need for human interference, or for assistance in high level reasoning tasks. Modern prominent examples of these are self driving cars. As such, most of the desire for intelligent systems, is one that could be used and be useful in the real world. For that, it needs to approach human level of intelligence.

Artificial Intelligence and Robotics, from fiction to reality



Figure 1.3: Marvin the paranoid robot ("The Hitchhiker's guide to the galaxy")

However, it is important to remind the high limitations of today's AIs. Indeed, reasoning akin to humans is expected, whereas only specialized intelligences have been developed in a real world context, as described by Yann Le Cun for the 2019 Turing award, in NYU's Mar 27, 2019 news release, where he predicts a revolution from specialized AIs to general AIs that approach human intelligence. And furthermore, the reasoning mechanism of an advanced artificial general intelligence (AGI) would probably be alien when compared to how humans reason due to the span of possible intelligences. Considering some works of fiction as simple thought experiments, some parallels with the current state of AI's can be observed that are concerning. For example, in the case where an AI is considered as fully competent as an AGI. Such as Nick Bostrom's thought experiment: "The paperclip maximizer", where an AGI is tasked with collecting as many paperclips as possible, and will try and get paperclips regardless of consequence to humanity or the world at large (which exemplifies the danger of oversimplified objective functions). Or more recently with specialized AI's that have dangerous edge cases, such as Tesla's & Uber's self driving car accidents, amazon's sexist recruitment tool, Microsoft's AI TAY which

when exposed to the internet quickly developed extremist points of view, or the French Chatbot that would suggest suicide (from AINews, October 28th 2020). Most of these issues can be narrowed down to alignment problems[5, 6], and reward gaming¹. As such, some concern should always be given when working with AI's, as the decision taken might not be logical or in line with all the principles that conceivably exist in human morality or what could be considered useful. Be it intentional for ill formed General intelligent systems, or unintentional for ill-trained specialized intelligences that could not grasp those concepts in a trivial manner.

Some historical perspectives regarding Artificial intelligence

The overall field of artificial intelligence can be dated back to 1940 with the first operational modern computer "Heath Robinson", and the foundational paper [1], both by Alan Turing. From these, the tools for creating intelligent system in theory and practice emerged as we know them today.

However the field of artificial intelligence is quite large, and spans expert systems, exploration & optimization algorithms, and machine learning. A brief overview of which can be seen in the

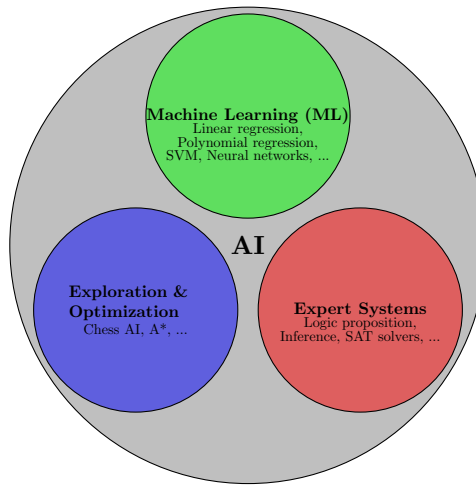


Figure 1.4: From *Artificial Intelligence: A Modern Approach* [7].

figure 1.4. The first class of methods are the expert systems, which suffer from combinatorial problems and definition issues when applied to controlling a robot, as there is often a continuous output to be controlled, and the rules that govern that output are often complex and/or opaque. This means that expert systems will be discounted as a viable solution in the thesis. The second class of methods are searching methods, similar to optimization algorithms or Chess AIs (e.g. alpha-beta decision tree search, A* search, ...). At first these could be considered a viable solution and indeed they are used in many cases (MDP controller, LQR control, ...). However they are often computationally expensive, and might not be able to solve some classes of problems (meaning some highly non linear and modal problems), due to the difficulty of efficiently searching the problem space. As such searching methods are also discounted as a viable solution in the thesis. The final class of methods are called machine learning methods (ML), which are methods that exploit computational statistics in order to generate "learned" models using some input data. They can be as simple as a regression algorithm, but in modern research it is more common to use deep artificial neural networks that are trained in a specific task [7, 8, 9].

An artificial neural network (NN), is a universal function approximator [10] that imitates the structure of neurons and synapses in brains. They were first suggested by Walter Pitts and Warren McCulloch in 1943 [11]. Marvin Minsky in 1951 then created the first single layer neural network called SNARC [12]. Then, after a few "AI-winters", Hopfield and Tank in 1985 realized that multilayer NN could solve optimization problems [13], with the back propagation algorithm being demonstrated in 1986 [14] (which is still widely used today), this laid the theoretical foundations for modern machine learning, which then continually expanded. Then in 2001, a large boom occurred with NN, as large datasets and improved computational power allowed for shifts in paradigm from focusing on the algorithm to focusing on the data. This in turn led to first GPU accelerated

¹A large collection of example can be seen here <https://tinyurl.com/56mh38b3>

NN called DAN CIRESAN NET [15], and then followed by AlexNet [16] scoring a new high score on image net, showing the potential of deep neural network architecture using GPUs in the early 2010's.

In the 1950s, the field of Reinforcement learning (RL) was also emerging from machine learning and control theory with Richard Bellman's work on dynamic programming approaches to design a controller that would minimize a measure of a system's behavior over time [17]. From this and from the works with neural networks lead to TD-gammon [18] in 1992. It was developed as the first neural network based method for reinforcement learning. Using the basis of TD-gammon and the developments in GPU accelerated deep neural networks, the DQN method was developed [19] that could play classic Atari games at human levels. Which lead to the methods such as AlphaGo [4], the first victory of an IA against an expert player at GO, and being able to solve a Rubik's cube with one hand as output and a camera as input [20].

AI applied to mobile robots

When observing the works done on AI's applied to mobile robotic control, much has been developed as early as 1988 with the ALVINN method [21] (shown in figure 1.5), that used a neural network to steer a car at 1.5km/h (due to computational constraints) in a supervised method. Followed by STANLEY [22] and BOSS [23] which both are self driving cars developed for the DARPA challenge, and both use classic control for the steering and speed, but use unsupervised and supervised machine learning for the computer vision and detection algorithms ². More modern approaches try and integrate neural networks for steering similarly to ALVINN, such as NVIDIA's end-to-end supervised driving [24] method using the camera input to steer the vehicle. Or using reinforcement learning approaches such as [25] which uses A3C [26] RL method to steer car, but this method is significantly below the supervised learning method, which imitates an expert. And also a Q-learning [19] approach [27] using discrete steering states, unfortunately it leads to oscillatory steering. And more recently the works on GT Sophy [28] where fundamental Markov issues have been addressed for training reinforcement learning methods with mobile robots.



Figure 1.5: The car ALVINN used.

Additional works have also been developed to improve the transferability of the method from the simulation where it is trained, so that its behavior remains consistent in real world conditions. Methods such as Sim-to-Real [29] by NVIDIA, or Intel's method for changing the simulated images to resemble reality [30]. As a results, end-to-end methods for autonomous driving are still ambitious.

Limitations of Control theory and AI applied to robotic control

When observing the limitations of control theory and AIs when applied to robotic control, an interesting paradox appears. For control theory, the key difficulties are adapting to the environment and autonomy, due to the complexity of the system's model that is needed in order to correctly adapt to the changes in the environment in an optimal manner. Where as, AI based robotic control struggles with robustness and predictability in its behavior, while often being able to adapt well to unseen conditions. From this, it seems that a hybridization of AIs and Control theory might lead to a more complete control scheme, one where a control law is able to exploit a known model for accurate control, all while the entire observation of the environment is taken into account by an AI, which can influence the control law in order to adapt to situation that would be unexpected when using the control law's simplified model.

²The use of AIs for detection and computer vision in robotics is a common theme, as it allows for safe controllability while allowing for intelligent detection, as such this approach is potentially sub-optimal from a control perspective.

1.2 Methods for adapting the control in complex environments using machine learning

The previous considerations tends to show that a high level of complementarity between deterministic approaches and machine learning techniques may open the way to new and efficient approaches. This is the subject of the proposed thesis, which may be formulated as follows:

Methods for adapting a control algorithm in complex environments using machine learning

Indeed this thesis was initiated in order to adapt the behavior of a mobile robot to variation in the perception quality and the variations of tire-ground interactions and the grip conditions, which would be difficult to achieve using control theory alone. However, adding AI's to a control law is not a singular task, and in reality is a continuous control spectrum, from full deterministic control law to full independent AI. As both extremes are sub optimal (as discussed previously and also verified in the following), there might exist a Goldilocks trade-off³ between the amount of control the AI and the control law have (and in which way) which could lead to a more adaptive control scheme than a single control law, while avoiding the limitation of the AI methods. This trade-off is what this thesis wants to explore and verify in order to determine whether such a hybrid method is indeed the best approach for mobile robotic control.

The context of the thesis

Advances in control and perception for mobile robotics have allowed the dissemination of mobile robotics for well known and well defined tasks. Be it in an indoor context such as production lines [31, 32], or in an outdoor context (autonomous mechanical weeding [33]), the mobile robots of today are capable of addressing specific, but restricted use cases in dynamic contexts. Indeed, most of the algorithms used today are designed to works with multiple sources of sensory information. As such, the absence of information or the use of noisy signals, renders a control law designed for a known precision less efficient. Even though many works exist for determining the quality of the perception [34, 35], the impact of said quality on the robotic behavior still remains underused, bringing the robot to halt, or to take improper movements.

In order to be able to exploit robotics in different evolving contexts and increase their autonomy, it seems then necessary to be able to adapt the behavior of the robot according to the perception quality as well as the varying motion parameters (velocity, grip conditions), which influence the robot dynamics. Indeed, in the presence of accurate localization information and known grip conditions, it would be sufficient for example to use adaptive and predictive controllers, allowing for a more accurate movement at higher speeds, despite the presence of dynamic phenomena. However, such approaches cannot be considered if the localization accuracy is degraded. In real world situation, such degradations in perception quality are common, due to a loss of GPS signal for example. These degradations bring then to a misinterpretation of the dynamics of the robot by the observation algorithms, leading to a source of potential instability. In addition, the tuning of control parameters depends on perception noise, and a set of control parameters may be stable with very accurate sensors and good grip conditions, but unstable when high noise or poor grip conditions are encountered.

Research axis

The subject of this thesis proposes the development of methods that allow the robot to consider the varying observations and the quality of perception. This should allow the adaption of the robot's performance to the perception, in order to preserve the stability and robustness of the robot. For this, the thesis will address the integration of the perception, as such as to adapt the control methods, their parameters, or certain control targets (forward speed, lateral distance to the trajectory, or others). The control modalities will be based on existing works, and their adaptation will be the first aspect of the thesis.

³Term derived from the fairy tale Goldilocks, where the titular character desired a balance that is "just right", not too "hot" and not too "cold".

The expected results, in the context of this subject, will be focused on the algorithmic adaptation mechanism. Demonstrating them in real world conditions will allow assessing the pertinence of the developments. These scenarios will be focused on the applications aimed for the agricultural domain, by considering the evolution of the off-road outdoor environment. The advances obtained during this thesis will allow robots to evolve as fast as possible, while guarantying a high level of precision and robustness, and consequently, safety aspects.

The integration of the varying observations and the quality of perception is a non trivial task. It would require significant changes to the modeling and controllers of used on the robots. These modifications are non ideal as they would be dependent on the robot and on the controller. Ideally, a method for automatically and correctly integrating the varying observations and the quality of perception into the control system could be considered. For this the field of machine learning could be leveraged in order to bridge the gap and reach an adaption of the robot's performance to the perception, in order to preserve the stability and robustness.

Implications of machine learning

When considering how to integrate machine learning methods, we must first distinguish what we wish to affect. Indeed most of the field of machine learning is based on outputting a value, that is based on the given input. The inputs are relatively easy to define, they are simply the varying observations and the quality of perception described previously. For the output however, it is not clear which element to affect; we can however draw a gradient from a pure classic control system, to a pure machine learning control system.

This can be interpreted as 4 levels of integration of machine learning in robotic control: The first and simplest, replace the controller with a machine learning method. The second, applying a corrective term to the existing controller using a machine learning method. The third, adjust the behavior of the controller through control parameters using a machine learning method. And the fourth, the trivial case of not using any machine learning.

Replacing the controller

Classically in machine learning applied to robotic control, when a control task is considered too complex for a control law, to directly control the robot using a machine learning method, such as the first self driving car using a neural network, called *ALVINN* [21]. Or for complex grasping or walking tasks [36].

Correcting control output

Considered an iterative improvement over replace an existing controller, is to correct the output of the controller, through the machine learning method. This allows a best of both worlds approach, where the machine learning method is capable of fully controlling the robot, but does not need to unless the controller has a sub-optimal behavior relative to an interior criterion of the machine learning method. In which case the machine learning method will correct the output in order to preserve the behavior that is desired.

These methods however can be difficult to implement, as they often depend on the machine learning method predicting and modeling on some level the behavior of the control law, in order to predict the control law and appropriately correct the control value.

Controller gains

Integrating the quality of perception and the perception into a control law is not a simple task, as it can be relatively hard to qualify how a controller should react to the perception information in full detail. Usually, controllers are tuned for a given environmental state, and this tuning encodes the unmodeled aspects of the controller. However, the quality of the perception changes this environmental state. An example of this would be GPS noise, as a higher GPS noise means the controller should reduce its settling time to the positional errors, as they are not as reliable. This tuning is defined as the gains of a controller.

The controller gains are values that define the reactivity of the controller to specific parameters. They are defined as the control effort relative to the error, in terms of time or distance to

convergence. They are usually set so the controller is critically damped, and so that the controller will quickly converge to the set point, but not overshoot too much or oscillate.

With this, an ideal gains value would then tune the controller to obtain a fast convergence to the set point, a non-oscillatory control, and to minimize the control errors overall.

Using a machine learning method, we could consider adjusting these predefined values in real time, in order to adapt the behavior of the robot to the environment.

Applying deep reinforcement learning to mobile robotics

The hypothesis we wish to prove is that a machine learning approach to control a mobile robot will be able to adapt its behavior in real time, as a function of the environment and the immediate state of the robot.

As such, the methods derived from machine learning will be compared over many trajectories and environment, in order to determine if the methods have been able to adapt the behavior of the robot, where existing controllers have not be able to do so. And to distinguish which method is the most appropriate for the use case tested in the thesis, which is mobile robots in an off road context.

From these works, an initial paper titled "*Neuroevolution with CMA-ES for Real-time Gain Tuning of a Car-like Robot Controller*" was published showing the promising results of these methods for gain tuning in simulation during the ICINCO 2019 conference, followed by a paper titled "*Online gain setting method for path tracking using CMA-ES: Application to off-road mobile robot control*" was published showing the success of the previous methods for gain tuning in real world conditions during the IROS 2020 conference. In order to analyze the results of these methods, an additional paper titled "*A Novel Gradient Feature Importance Method for Neural Networks: An Application to Controller Gain Tuning for Mobile Robots*" was published demonstrating a method for analyzing the influencing inputs of a neural network using a gradient approach during the ICINCO 2020 conference; of which it was extended with an in-depth analysis of the methods previously shown for the LNEE journal. During the PhD an internship was achieved in order to determine if tuning the speed instead of the control gains was a viable research path to explore, the results of which lead to the paper titled "*Online velocity fluctuation of off-road wheeled mobile robots: A reinforcement learning approach*" shown in the ICRA 2021 conference. And finally, a paper titled "*On the online tuning of control parameters for off-road mobile robots: Novel deterministic & neural network based approaches*" where the *model gain tuner* and the dynamic parameters are taken into account for gain tuning at $4m.s^{-1}$ were developed for the RAM journal which was accepted, which also lead to a presentation of the works during the ICRA 2022 conference as a guest. Overall a total of 4 conference papers, 2 journal papers, and 1 patent were developed during the PhD, along with two more journal paper to be submitted.

This thesis manuscript is structured as follows: First, a summary of robotic control, robot modeling, and the simulation built for training the method is described in order to clarify the notions used throughout the thesis. Next, a state of the art of deep reinforcement learning for robotic control is described, in order to show which method is the most appropriate for this task, and to explain how these methods can be used for adapting the behavior of the robot. This is followed by a comparison of the existing controllers with the machine learning steering method, the machine learning corrective steering method, and the machine learning gain tuning method. This is then compared against a new model based gain tuning method and tested with dynamic parameters as additional inputs. Then, a hybrid speed and steering control is proposed, in order to adapt the full behavior of the robot to the environment. In conclusion, the key aspects discovered along the way for the machine learning methods are outlined, and also the current existing limitations that could be pushed even further in order to improve certain aspects of the thesis if addressed in future works. Throughout the thesis, additional literature will be referenced as needed, in order to introduce key notions as needed, and to streamline the flow.

Chapter 2

Vehicle Modeling and control

Classically, the navigation for autonomous vehicles is determined using a control law, which assures the correction of the tracking errors from the referenced trajectory. Among all the possible solutions to this, here we have chosen to present in particular two model predictive control law (MPC), along with their kinematic model they are based on. The dynamic model presented is used for simulation purposes, in order to generate a training environment that is realistic enough, as well as for the derivation of an observer for adapting grip conditions. Furthermore, this dynamic model will be used to define the *Model gain tuner* described further on, and in order to help define the observer needed in the future sections.

2.1 General features about modeling

The design of control laws for mobile robot autonomous navigation basically relies on the use on a model allowing to characterize the robot motion with respect to control variables [37, 38]. In the framework of this thesis, and without loss of genericity, the control of car-like mobile robot is considered in an off-road environment, acting at relatively high speed. As a result, the classical use of pure kinematic model [39], relying on rolling without sliding assumption cannot be applied in this context, since many disturbing phenomenon cannot be neglected, such as effect of bad grip conditions and dynamical phenomenon. As a result, more complex modeling must be handled to account for such effect in order to derive an accurate control, despite facing such perturbations.

2.2 Kinematic Model

Assumption and notations

The first level of modeling considered for mobile robot control in this work, consists in considering only the speed vectors, and ignoring the more complex dynamic behavior of the robot. This simplification assumes no sliding dynamic between the wheels and the ground.

The Ackermann-type front-steering vehicle is modeled in the yaw plane as a bicycle [39], the front and rear axles being modeled by a single wheel, and the steering angle as an equivalent angle. A model of the kinematics of this system may be sufficient depending on the context; the more complex dynamic behavior of the robot is then neglected. In this case, the assumptions of rolling without longitudinal and lateral slippage are applied. Using such an assumption, and considering that all wheels stay in contact with the ground, one can represent mobile robot motion such as on the figure 2.1

In this representation, the middle of the rear axle is used as the reference point, the position of which, has to be controlled. In the framework of path tracking problem, the robot state is defined with respect to a desired trajectory (D), to be followed by this reference point. In order to compute the motion model, we use the following notations:

- (D) is the trajectory followed by the robot, represented as a geometric curve.
- s is the curvilinear abscissa along (D), at the closest point belonging to the reference trajectory from the robot reference point.

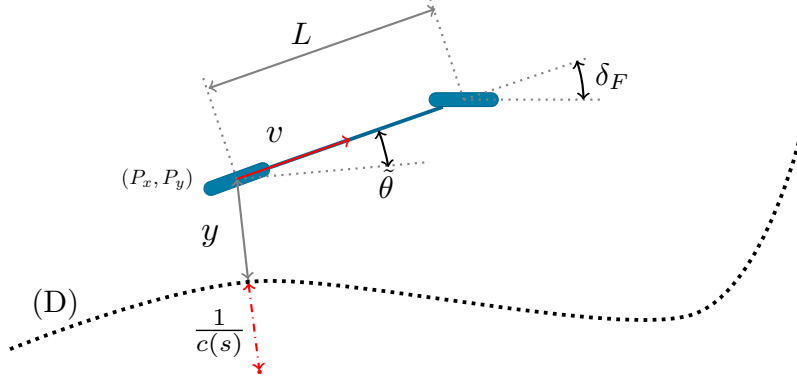


Figure 2.1: The cinematic robot model.

- L is the wheel base length of the robot.
- v is the speed vector of the robot at the middle of the rear axle.
- P_x, P_y is the position of the midpoint of the rear axle in a global reference frame.
- θ is the robot's heading, in a global reference frame.
- $\tilde{\theta}$ is the angular error, defined as the angular difference between the robot's heading and the tangential angle at the nearest point along the curvilinear abscissa of the trajectory.
- y is the lateral error, defined as the distance of the rear axle to the nearest point along the curvilinear abscissa of the trajectory.
- $c(s)$ is the curvature of the point of curvilinear abscissa s on the trajectory.
- δ_F is the front steering angle.
- a is the robot longitudinal acceleration.

With the assumption of no lateral movement of the wheels, the speed vector orientations are given by the front and rear wheels directions. Under this assumption and considering there is no instantaneous change in the speed, the following system of equations can be derived as (see [40] for details):

$$\begin{cases} \dot{P}_x &= v \cos(\theta) \\ \dot{P}_y &= v \sin(\theta) \\ \dot{\theta} &= v \frac{\tan(\delta_F)}{L} \end{cases} \quad (2.1)$$

When in the Frenet frame projection, this model can be rewritten as follows:

$$\begin{cases} \dot{s} &= v \frac{\cos(\tilde{\theta})}{1 - c(s)y} \\ \dot{y} &= v \sin(\tilde{\theta}) \\ \dot{\tilde{\theta}} &= v \left[\frac{\tan(\delta_F)}{L} - \frac{c(s) \cos(\tilde{\theta})}{1 - c(s)y} \right] \end{cases} \quad (2.2)$$

One can note that this model has a singularity at $y = \frac{1}{c(s)}$ (the robot is situated at the center of the reference path curvature), however this is unlikely to happen, due the how large $\frac{1}{c(s)}$ is, with respect to y .

2.3 Dynamic model

Improving the Kinematic model

In order to account for bad grip conditions, one can consider a second layer of modeling, based on dynamical model. Since it has a great interest in automotive industry, many works have been

achieved on the study of car dynamics (sec [41]). In the framework of autonomous navigation, many approaches using dynamical model keep the bicycle representation, in order to reduce the number of parameters to be known [42]. Since this work aims at considering the development of IA approaches to autonomously tune the control parameters of a deterministic control law, a dynamical model is here exploited to observe and simulate the robot behavior. For this reason, we consider the simplest dynamical model (with as less parameters as possible) shown in [43], here applied to a single steering axle.

In this representation the longitudinal position of the center of gravity has to be known, as well as the robot mass m and the moment of inertia I_z along the vertical axis. The motion is supposed to be achieved on a flat ground, avoiding the consideration of a bank angle. Moreover, since we consider that the velocity is changing slowly, the longitudinal motion is not considered. As a result the contact forces acting at the tire ground patch are considered to be only oriented along a perpendicular axis with respect to tire's directions

Using these assumptions, the description of robot motion can be described as depicted on the figure 2.2:

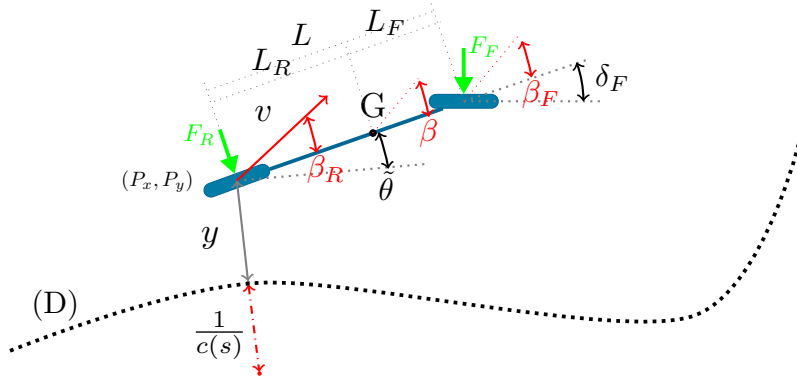


Figure 2.2: The dynamic robot model.

Similarly to the kinematic model, the rear axle midpoint is used as the reference for the speed, lateral error, angular error, and position measurement. Notations are defined as follows:

- G is the center of mass of the robot.
- L_R and L_F are the distance from the center of mass to the rear and front axle respectively.
- F_R and F_F are the lateral force on the rear and front axle respectively.
- β is the vehicle sliding angle.
- β_F and β_R are the front and rear axle sliding angle respectively.
- I_z is the moment of inertia across the Z axis (here meaning away from the ground).
- v_2 is the longitudinal component of the speed vector v .

In this point of view, the robot is supposed to move at an almost constant speed. From this modeling, the following system of equations can be derived:

$$\begin{cases} \ddot{\theta} &= \frac{1}{I_z} (-L_F F_F \cos(\delta_F) + L_R F_R) \\ \dot{\beta} &= -\frac{1}{v_2 m} (F_F \cos(\beta - \delta_F) + F_R \cos(\beta)) - \dot{\theta} \\ \beta_R &= \arctan(\tan \beta - \frac{L_R \dot{\theta}}{v_2 \cos(\beta)}) \\ \beta_F &= \arctan(\tan \beta + \frac{L_F \dot{\theta}}{v_2 \cos(\beta)}) - \delta_F \\ v_2 &= \frac{v \cos(\beta_R)}{\cos(\beta)} \end{cases} \quad (2.3)$$

Tyre slip model

In order to model the lateral forces F_F and F_R applied to the dynamic model, a tyre slip model must be used. A tyre slip model, is a modeling of the tyre dynamic that predicts the lateral forces applied to the wheels, based on the systems current state. A trivial example of this is a linear approximation of the lateral forces F_F, F_R with respect to the tyre slip angles β_F, β_R

$$\begin{cases} F_F &= C_F \beta_F \\ F_R &= C_R \beta_R \end{cases} \quad (2.4)$$

where C_F and C_R are the front and rear cornering stiffnesses respectively. A higher C_F, C_R implies a stronger force while cornering, which implies better grip conditions. This modeling is not optimal however in many cases, due to a lack of saturation of the lateral forces which occurs when the sliding angles are high enough (this is due to the lateral force matching or exceeding the maximal friction force of the tyre), this situation of force saturation often occurs in an off-road context, which is the context studied in our case [44].

As such, we need to consider a better modeling of the tyre's lateral forces. For this, many modeling methods exist, such as TMEASY [45] and LuGre [46] (for a more in depth look of tyre slip models, refer to [47]). For our use case, the famous Pacejka tyre slip model [48] was initially chosen, due to its modeling accuracy, low number of parameters, and modeling of the desired force saturation.

From the Pacejka tyre model, the force equation F_y is described as:

$$\begin{cases} F_y &= D \sin(C \arctan(B\phi)) \\ \phi &= (1 - E)\alpha + \frac{E}{B} \arctan(B\alpha) \\ D &= a_1 F_z^2 + a_2 F_z \\ B &= \frac{a_3 \sin(a_4 \arctan(a_5 F_z))}{CD} \\ E &= a_6 F_z^2 + a_7 F_z + a_8 \end{cases} \quad (2.5)$$

Where a_1 to a_8 , and C describe the properties of the tyre and the ground. F_z is the vertical force applied to tyre and β is the tyre side slip angle.

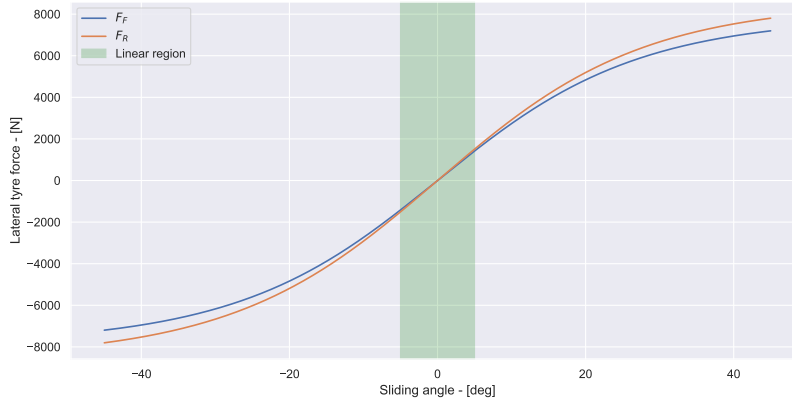


Figure 2.3: The Pacejka model's lateral force curve, using a $F_z = 1.055\text{kN}$, with respect to the tyre slip angle.

An example of the results of this equation is visible on the figure 2.3. One can notice that this model is non linear, despite a pseudo linear part for low sideslip angles (within $\pm 5^\circ$) corresponding to the previous linear model (2.4). This so-called pseudo-sliding part is often used for autonomous vehicle control moving at low speed in urban environment. Above this part, the lateral forces tend to saturate after having reached a maximum value. In such areas, sliding is effective and the assumption of pure rolling without sliding for the wheel (used to obtain the model (2.1)) is no longer valid.

However, it was discovered in sec 5, that the Pacejka tyre slip model was difficult to align to real world conditions leading to a large high systemic error between the simulation and real world environment. As such, the final tyre slip model used is a Pacejka tyre slip model with a very large

linear region, similarly to the model described in (2.4), as it is easily set using an observer from real world experiments.

Actuators delays

For modeling purposes in mobile robotics, actuators delay can be divided into two types: a pure delay on the signal and a n-th order delay simulating the physics of the actuator. The pure signal delay models the transmission and electronic delay from the control system to the actuator. It is simulated by using an array of values storing the last n steps, and loading the n-th value as the output value. The n-th order delay models the physical aspects of maximal forces, speeds, and inertia of the actuator system. It is simulated by passing the command signal into dynamic equations that represent a n-th order delay equation, which is solved through the simulator's integrator, and from that the simulated steering angle is resolved.

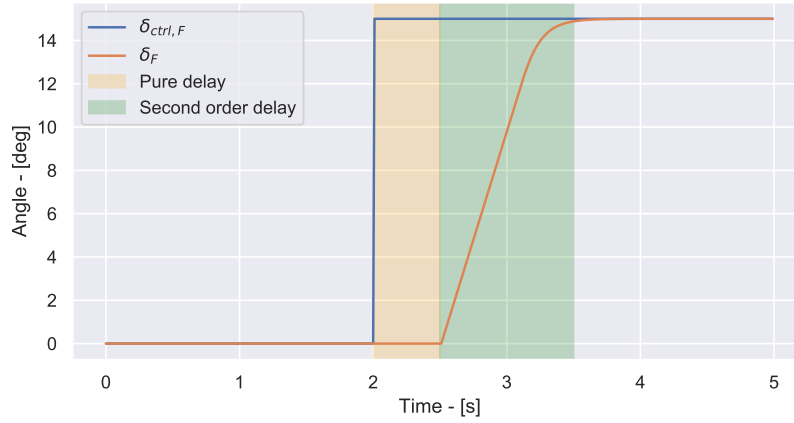


Figure 2.4: An example of the model's delayed steering

Figure 2.4 depicts an example of the steering delay that will be used in the following. In the yellow section, the pure action delay can be observed, where the control output δ_f does not react for 0.5s. In the green section, a second order delay can be observed succeeding the pure delay, the control output starts to converge towards the desired value at a constant rate, and then slows down exponentially when the target value start to be reached. ¹

From these two delay modeling in conjunction, we can now approximate the steering behavior of the mobile robot, in a simulated context.

2.4 Extended kinematic model

In order to derive a controller for the mobile robot, the dynamical model introduced in (2.3) is not very tractable due to the number of parameters that need to be known prior. Parameters such as the mass and inertia can be determined easily, but the varying grip conditions encountered in off-road applications would require a fast and accurate estimation of said parameters. Despite control laws based on dynamical model are possible [49], we considered in this thesis the application of a control algorithm based on an alternative motion representation. Indeed we can consider the kinematic interpretation of the sideslip angles (used to derive contact forces), in the purely kinematic representation depicted on the figure 2.5. This "extended kinematic model" [50] consists in considering the actual orientation of speed vectors at each contact point, which differs from the tire's directions by an angle β . As pointed out on the figure 2.5, two sideslip angles have to be considered: β_F and β_R for the front and rear wheel respectively.

Thanks to this representation, we can derive the motion equation in the same way as for a purely kinematic model. As detailed in [51], the motion equation of the extended model may be computed as:

¹Eq derived from the second order equation, with a time constant τ_s : $\tau_s^2 \frac{\partial^2 y}{\partial t^2} + 2\tau_s \frac{\partial y}{\partial t} + y = 0$, in order to define:

$$\frac{\partial^2 \delta_f}{\partial t^2} = \frac{u_{\delta_f} - \delta_f}{\tau_s^2} - \frac{\partial \delta_f}{\partial t} \frac{2}{\tau_s}$$

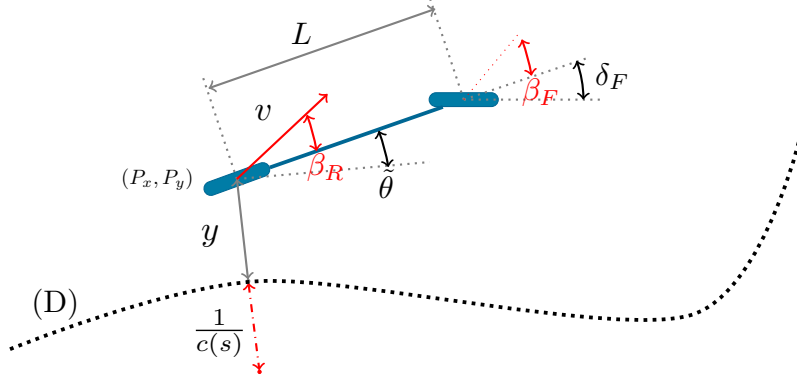


Figure 2.5: The extended cinematic robot model.

$$\begin{cases} \dot{P}_x &= v \cos(\theta + \beta_R) \\ \dot{P}_y &= v \sin(\theta + \beta_R) \\ \dot{\theta} &= v \frac{\tan(\delta_F + \beta_F)}{L} \end{cases} \quad (2.6)$$

When in the Frenet frame projection, this model can be rewritten as follows:

$$\begin{cases} \dot{s} &= v \frac{\cos(\tilde{\theta} + \beta_R)}{1 - c(s)y} \\ \dot{y} &= v \sin(\tilde{\theta} + \beta_R) \\ \dot{\tilde{\theta}} &= v \left[\frac{\tan(\delta_F + \beta_F)}{L} - \frac{c(s) \cos(\tilde{\theta} + \beta_R)}{1 - c(s)y} \right] \end{cases} \quad (2.7)$$

As such, an extended kinematic model has similar properties when compared to a classical kinematic model, and as such similar classical control methods can then be applied.

2.5 Deterministic steering control

As the thesis is based on the adaptation of control laws, one has to derive a control strategy for trajectory tracking of a mobile robot. The application context of off-road motion suggests to account for bad grip conditions. For this, control laws that can account for the sliding angles of the robot and that are predictive are of interest, as they will be strong candidates for starting point from which we can compare the methods developed in the following sections.

Adaptive control law from a chained system [Romea]

The first control law is an existing and proven predictive controller for agricultural robotic control, that uses a chained system as a base for deriving its control equation. The following will briefly describe this derivation, but a more detailed explanation can be found from [51].

Using as a basis the Frenet kinematic equations (2.2) that are extended with the sliding angles (as shown in equations 2.7), the following chained system can be derived:

$$[s, y, \tilde{\theta}] = [a_1, a_2, a_3] = [s, y, (1 - c(s)y) \tan(\tilde{\theta} + \beta^R)] \quad (2.8)$$

with the following control variables:

$$[v, \delta_F] = [m_1, m_2] = \left[\frac{v \cos(\tilde{\theta} + \beta^R)}{1 - c(s)y}, \frac{\partial a_3}{\partial t} \right] \quad (2.9)$$

Deriving the chained system $[a_1, a_2, a_3]$ over the curvilinear abscissa denoted s (or a_1), the following system is obtained:

$$\begin{cases} \frac{\partial a_1}{\partial a_1} &= 1 \\ \frac{\partial a_2}{\partial a_1} &= a_3 \\ \frac{\partial a_3}{\partial a_1} &= m_3 \end{cases} \quad (2.10)$$

Where $m_3 = \frac{m_2}{m_1}$ is a new control variable relying on the steering angle. In order to ensure the convergence of the tracking error $y = a_2$ to zero, a judicious choice of m_3 can be:

$$m_3 = -K_d a_3 - K_p a_2 \quad (2.11)$$

by injecting this expression into the system (2.10) indeed leads to this second order differential equation for a_2 :

$$y'' + K_d y' + K_p y = 0 \quad (2.12)$$

Since K_p, K_d are manually chosen, one can select such parameters, which are homogeneous with proportional and derivative gains, to obtain a stable convergence of tracking error to zero. Using the reverse transformations from (2.8) and (2.9), one can finally obtain the explicit control expression for front steering angle:

$$\delta_F = \arctan \left(\tan(\beta_R) + \frac{L}{\cos(\beta_R)} \left[\frac{c(s) \cos(\tilde{\theta}_1)}{\alpha} + \frac{A \cos^3(\tilde{\theta}_1)}{\alpha^2} \right] \right) - \beta_F \quad (2.13)$$

with:

$$\begin{cases} \tilde{\theta}_1 &= \tilde{\theta} + \beta_R \\ \alpha &= 1 - c(s)y \\ A &= -K_p y - K_d \alpha \tan(\tilde{\theta}_1) + c(s) \alpha \tan^2(\tilde{\theta}_1) \end{cases}$$

This control expression constitutes the expressions to be applied on the steering angle to ensure the differential equation (2.12) for tracking error. It requires the knowledge of sideslip angles than can be observed from the robot's state, as demonstrated in [51] and briefly detailed in section 2.7. One has then to substitute the actual value for sideslip angles β_F and β_R their equivalent observed values $\hat{\beta}^F$ and $\hat{\beta}^R$ in control expression (2.13).

Since this control law is only reactive, it does not account for actuator delays, low level reaction times, as well as inertial effects that affect the settling time for the robot to reach a desired yaw rate. Also, variation in the curvature of the trajectory has to be handled. As such, a predictive method is employed that corrects the desired steering angle, in order to follow the reference trajectory, while taking into account the expected steering behavior over a time horizon.

For that, we consider the non-zero part of the left term in equation 2.13 in the absence of tracking errors and slippage, and replace it with a corrective term, based on a predictive curvature method that is detailed in [51]. It depends on a time horizon parameter H , which defines the distance of the curvature lookahead in seconds [s].

predictive control law from constraint optimization [EBSF]

The second control law is also an existing and proven predictive controller for mobile robots, but over a distance horizon instead of a time horizon, and under constraints. The following will briefly describe how it functions, but a more detailed explanation can be found from [52].

It is based on constrained linear optimization, where the constraint is formalized such that the front and rear ends of the mobile robot must remain within a corridor around the reference trajectory defined by lateral tolerance values. This control law's inputs are derived by solving the following system of equations:

$$\begin{cases} \min_U \frac{1}{2} \mathbf{U}^t (\mathcal{B}^t \mathcal{Q} \mathcal{B} + \mathcal{R}) \mathbf{U} + \mathbf{y}_0^t \mathcal{A}^t \mathcal{Q} \mathcal{B} \mathbf{U} \\ \mathcal{D} \mathcal{B} \mathbf{U} + \mathfrak{d}_{gap} + \mathcal{D} \mathcal{A} \mathbf{y}_0 \geq \mathbf{0}_{4n \times 1} \end{cases} \quad (2.14)$$

Where $\mathbf{U} \in \mathbb{R}^n$ is the control inputs vector at each of the n steps over the chosen distance horizon, $\mathbf{y}_0 \in \mathbb{R}^3$ is the current error state vector, $\mathcal{A} \in \mathbb{R}^{3n \times 3}$ and $\mathcal{B} \in \mathbb{R}^{3n \times n}$ are the matrices of the system linearized predicted states $\mathbf{Y} = \mathcal{A} \mathbf{y}_0 + \mathcal{B} \mathbf{U}$ with $\mathbf{Y} \in \mathbb{R}^{3n \times 1}$ the predicted error state vectors, $\mathcal{D} \in \mathbb{R}^{4n \times 3n}$ is a block diagonal matrix that contains the dimensions of the robot represented as a rectangle, $\mathfrak{d}_{gap} \in \mathbb{R}^{4n \times 1}$ is a vector of the gap size that the constraint must respect, and $\mathcal{Q} \in \mathbb{R}^{3n \times 3n}$ and $\mathcal{R} \in \mathbb{R}^{n \times n}$ are block diagonal gain matrices with elements $\gamma_Q^k Q$ and $\gamma_R^k R$ respectively. $(\gamma_Q; \gamma_R) \in]0; 1]^2$ are forgetting factors, $k \in [1; n]$ is the index of a block in the diagonal, and $Q \in \mathbb{R}^{3 \times 3}$ and $R \in \mathbb{R}^{1 \times 1}$ are positive definite chosen matrices.

The minimization equation can be split into two parts, the first $\frac{1}{2}\mathbf{U}^t(\mathcal{B}^t\mathcal{Q}\mathcal{B} + \mathcal{R})\mathbf{U}$ minimizes the control energy, and the second $\mathbf{y}_0^t\mathcal{A}^t\mathcal{Q}\mathcal{B}\mathbf{U}$ minimizes the lateral error over the predicted horizon. From this, the optimizer finds the vector \mathbf{U} that minimizes the control energy and the lateral error over the predicted horizon, while respecting the constraint.

Predictive control parameters are $\mathcal{Q}(0, 0)$, $\mathcal{Q}(1, 1)$, $\mathcal{Q}(2, 2)$, and $\mathcal{R} = 1.0$. Parameters of the diagonal of \mathcal{Q} that define the tracking efficiency refer to the lateral error and its two successive derivatives with respect to the curvilinear abscissa. Parameter of \mathcal{R} which defines the tracking smoothness refers to the variation in front axle steering relative to the curvilinear abscissa.

The spatial window S is the discretization step, here chosen at 10 cm for an accurate model. Decreasing it increases the accuracy but also the calculations. Similarly, $n = 20$ is the number of steps chosen to define the prediction horizon $nS = 2m$. The prediction horizon is chosen greater than the system settling time ($2.2ms^{-1}$ max speed $\times 0.15s$ steering response time = $0.33m$), and large enough to anticipate the trajectory, but not too large to remain within the validity domain of the model. It has been successfully used in existing projects, and was chosen as it is fundamentally different approach when compared to the previous control law, this allows for a more in-depth analysis of our results in the following sections, as it will show if any of the developed methods are agnostic or dependent on type of control law used.

Tuning control laws parameters

These control laws have been shown in experimentations to be robust, reliable and predictable. However, they do require some expert knowledge of the system, in order to correctly tune their control gains for a given situation², which might be dependent on numerous varying parameters, which makes then difficult to estimate in real time. Furthermore, these control laws depend on strong hypotheses and simplifications, which become problematic in strongly dynamic environment, where these hypotheses and simplifications no longer hold true. For instance, in the first control approach, three parameters have mainly to be tuned:

- K_p : analogous to a proportional gain, it mainly defines the theoretical distance of convergence with respect to the lateral error.
- K_d : analogous to a derivative gain, it defines the theoretical distance of convergence with respect to the angular deviation.
- H : the time horizon of the predicted curvature, allowing the controller to compensate for the convergence time of the steering with respect to the curvature.

The performance and the stability of the path tracking is highly dependent on the choice of these parameters, which define the settling distance for the lateral error, as well as an anticipation windows. Ideally, these parameters need to be tuned to react as fast as possible, all while preserving the stability (behavior without oscillations). Such an optimal behavior is not easy to obtain, as it depends on numerous parameters, such as the actuators, the velocity, the sensor noises, and the grip conditions. To highlight this fact, let us consider simulations using the test-bed hereafter detailed with two set of parameters:

- set 1 Settling Distance (SD=5m), $H = 0.5s$
- set 2 Settling Distance (SD=15m), $H = 0.5s$

The robot follows a desired trajectory at 3 m/s composed of a straight line, a constant curve, and a second straight line on two kinds of grip conditions defined by two sets of parameters in the Pacejka model :

- GripA - equivalent to asphalt
- GripB - equivalent to mud

²Due to the non-linear system, transfer function analysis and pole finding is not applicable.

Results on tracking error, obtained during the path following with two sets of parameters and two kinds of grip conditions

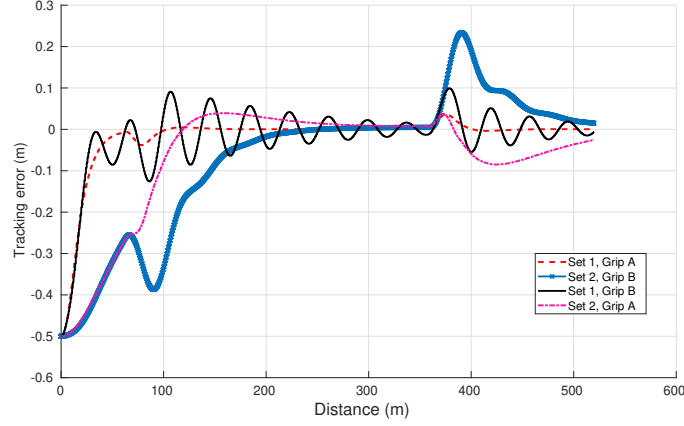


Figure 2.6: An example of the model's delayed steering

One can see on this figure the set 1 allows a fast convergence when grip conditions are good (asphalt), but starts to be unstable with low grip conditions are encountered despite sideslip angles being correctly estimated (oscillation around null tracking). On the contrary, the second set for control parameters leads to a stable behavior even when grip conditions are bad, but the settling distance is sub-optimal when grip conditions are good. The same conclusions can be obtained with respect to the velocity, actuator delays, or sensor noises.

This example shows the difficulty of tuning a constant parameters with respect to uncertain and dynamic environment, as well as the limitation of using constant parameters, that by definition are not optimal in all conditions.

However, there have been some considerable advances in the field of artificial intelligence in last 5 years, specifically in machine learning and reinforcement learning. So, a natural question would be, is it possible to replace existing control laws using this field of computer science? Indeed, it has been very successful in many control tasks, such as solving a Rubik's cube with one hand [20]. This has shown its adaptability and performance when compared to existing control laws. However these methods are not foolproof, as they are likely less stable and predictable due to black box nature of some of the methods used, and they require learning from scratch, which ignores the field of robotic control and the field of automation in general.

2.6 Extended Kalman filter

In the following, an extended Kalman filter [53] is used as state estimator. A Kalman filter consists of two steps; the prediction step and the correction step. The prediction stage: The model and inputs are used to predict the behavior of the robot. This phase is formulated as such for an extended Kalman filter:

$$\begin{aligned}\hat{x}_{k+1} &= f(\hat{x}_k, u_k) \\ \mathcal{C}_k &= F_k \mathcal{C}_k F_k^T + Q\end{aligned}\tag{2.15}$$

where \hat{x}_{k+1} is the predicted value of the state, f is the non-linear extended kinematic (see section 2.4) model of the robot, and F_k is the Jacobian matrix of f expressed as follows:

$$F_k = \left. \frac{\partial f}{\partial x} \right|_{\hat{x}_k, u_k}$$

The Correction stage: A measurement is used to correct the predicted value of the state. This phase is described by:

$$\begin{aligned}\hat{x}'_k &= \hat{x}_k + K'(z_k - h(\hat{x}_k)) \\ \mathcal{C}'_k &= \mathcal{C}_k - K' H_k \mathcal{C}_k \\ K' &= \mathcal{C}_k H_k^T (H_k \mathcal{C}_k H_k^T + R_k)^{-1}\end{aligned}\tag{2.16}$$

where \hat{x}'_k is the estimated value of the state. h is the measurement function; which describes how the measurements are related to the internal state. H_k is the Jacobian matrix of h :

$$H_k = \left. \frac{\partial h}{\partial x} \right|_{\hat{x}_k, u_k}$$

Q, R , and \mathcal{C} are covariance matrices of the model, system noise, and corrected estimation respectively. Q is fix, and represents the errors in the given robot model, R is dynamic and it changes depending on the level of noise in the measurements, \mathcal{C} combines the two and is updated every timestep.

2.7 Observers

In control theory, some parameters are needed in order to compute the optimal steering angle, using the controller's internal model of the environment. Some kinematic or dynamic parameters are not trivially measured from the robot's sensors, as such observers are used in order to estimate their values. Observers consists of ideal models of the robot and its environment, from which a predicted state is determined and compared to the measured state, and from which the observed parameter is determined.

In this work, two observers are used, the first is a kinematic observer that derives the sliding angles from the robot's estimated state, the second is a dynamic observer that derives the cornering stiffnesses from the robot's estimated state and the estimated sliding angles.

Sliding angles observer

As described previously, the sliding angles (in $[rad]$) determine the difference between the steering state's, and the robot's rate of change it's heading (i.e. the effective steering state). These angles are directly proportional to the lateral force over the wheels of the robot, as such it is paramount in order to effectively determine and model the steering's behavior over time.

As detailed in [51], it is derived from the following state equation:

$$\dot{x} = \begin{bmatrix} \dot{x}_{pos} \\ \dot{x}_\beta \end{bmatrix} \quad (2.17)$$

Where $x_{pos} = [P_x \ P_y \ \theta]^T$ (the x, y positions, and heading) and $x_\beta = [\beta_F \ \beta_R]^T$ (the front and rear sliding angles). Which is then used with the function $f(x, v, \delta)$:

$$f(x, \delta_F, v) = \begin{bmatrix} v \cos(\theta + \beta_R) \\ v \sin(\theta + \beta_R) \\ v \cos(\beta_R) \frac{\tan(\delta_F + \beta_F) - \tan(\beta_R)}{L} \end{bmatrix} \quad (2.18)$$

In order to derive the observer:

$$\dot{\hat{x}} = \begin{bmatrix} \dot{\hat{x}}_{pos} \\ \dot{\hat{x}}_\beta \end{bmatrix} = \begin{bmatrix} f(x_{pos}, \hat{x}_\beta, \delta_F, v) + \alpha_{pos} \\ \alpha_\beta \end{bmatrix} \quad (2.19)$$

Where $\hat{x} = [\hat{x}_{pos} \ \hat{x}_\beta]^T$ are the observed sliding angles. In order to convergence $x - \hat{x}$ towards zero, the following definition is given:

$$\begin{cases} \alpha_{pos} &= K_{pos} \tilde{x}_{pos} \\ \alpha_\beta &= K_\beta \left[\frac{\partial f}{\partial x_\beta}(x_{pos}, \hat{x}_\beta, \delta_F, v) \right]^T \tilde{x}_{pos} \end{cases} \quad (2.20)$$

Where $\tilde{x}_{pos} = x_{pos} - \hat{x}_{pos}$ is the observation error, and K_{pos} and K_β are tunable parameters. From this, \hat{x} can be calculated, which derives the desired observed values.

This method however has some limitations, such as the derivatives of the sideslip angles are considered null, and the derivation from a kinematic model instead of a dynamic model.

Cornering stiffness observer

As described in previously, the cornering stiffnesses (in $[kN.rad^{-1}]$) determines the relation between the sliding angle and the lateral force over the wheels of the robot. This stiffness is related to the characteristics of the robot's wheels, the robot's mass, and the grip conditions of the ground. As such estimating these parameters allow for a rough approximation of the grip conditions, which helps predict the behavior of the robot with respect to the grip conditions.

The first step of this observer is to derive the sliding angles, which can be achieved using the previous observer.

The second step consists of the following dynamic model:

$$A(x) = \begin{bmatrix} 0 \\ -\dot{\theta} + \frac{g \sin(\alpha)}{v} \end{bmatrix}, \quad B(\delta_F, \delta_R) = \begin{bmatrix} \frac{-L_F \cos(\delta_F)}{\frac{I_z}{vm}} & \frac{L_R \cos(\delta_R)}{\frac{I_z}{vm}} \\ \frac{-\cos(\delta_F)}{vm} & \frac{\cos(\delta_R)}{vm} \end{bmatrix} \quad (2.21)$$

With the state vector $x = [\dot{\theta}, \beta]^T$.

From this an observed state \hat{x} is defined, where exponential convergence method allows for the convergence towards zero of $x - \hat{x}$.

Once \hat{x} is derived, the lateral forces can be inferred, and from those using a linear cornering stiffness model ($F_F = C_F \beta_F$ & $F_R = C_R \beta_R$), the final C_R and C_F cornering stiffnesses are obtained.

This observer has some limitation however, such as the loss of observability when the sliding angle are too small, and the use of a linear cornering stiffness model which starts to become inaccurate when the sliding angles are too large. In practice, the cornering stiffness is frozen if the sliding angles are below a given threshold, in order to avoid erroneous observations.

2.8 Simulated implementation of the models and controllers

Accurate and deterministic

The simulator plays an important role in the training and validation process for robotic control methods. As it must be accurate enough in order to correctly estimate the result of the differential equations defined by the model. And be deterministic in order for the results, training, and experiments to be repeatable. This simulator is fully written in approximately 9000 lines of *C++* code, and is the result of the source code being rewritten three times from scratch, with an emphasis on readability first, modifiable second, and performance third (details of which are given in appendix A.10).

In order for the simulator to be representative enough, the dynamical set of equation (2.3) is computed to take into account for the influence of dynamical effect as well as low grip conditions, together with a second order model to account for actuator delays. A sampling rate of $\Delta t = 0.01s$ was used, with a fourth-order Runge-Kutter [54] ordinary differential equation (ODE) solver. As an Euler ODE solver was unstable with the sliding effect of the dynamic model. Furthermore, the simulator is designed so that the ODE solver can be easily changed, allowing for quick experimentation with higher order ODE solvers. It is important to note that adaptive ODE solvers were tested on the simulation (Runge-Kutter-Fehlberg 45 [55] and Adaptive Dormand-Prince 56 [56]) with the error margin set to a value that would lead to comparable compute times to Runge-Kutter 4, however they cause the simulation to underestimate the lateral forces, leading to a significant error in the accuracy, which was observed when a trained model using this method, performed very poorly in real world conditions.

In order to make the simulator deterministic, a pseudo random number generator set with a known seed, is used for each parallel thread. This means that the undeterministic nature of operating system's process scheduling is nullified, making the training process fully repeatable. Furthermore, a Mersenne Twister pseudo random number generator is used, and can be easily changed in the simulator. Careful attention should be made with the Mersenne Twister pseudo random number generator, as it has shown to be sub-optimal if the seeding is poorly defined. An alternative that does not suffer from this problem is the XorShift64.

Computationally fast

The computing speed is a key aspect in any simulation, and becomes a strong aspect in the case of using evolutionary strategies, where a year of simulated time, must be calculated in a reasonable amount of time.

For this, the simulation is completely written in *C++*, with some visualization tools, written in python, that load the *C++* generated csv files. However, simply writing in *C++* will not yield the optimal performance, the code must be benchmarked and analyzed, so that useful optimization can be added. The benchmarking tool *perf* was used, which count the CPU cycles spent in a given function, this allows a good visualization of where the bottleneck in performance is. From this, here are the non-trivial optimizations that were added to the simulation:

- *Eigen* library: The hand implementation of matrix multiplication is a non-trivial task, as it requires the developers to write the loops in a cache efficient manner, while also taking into account possible hardware acceleration such as *AVX*. *Eigen* is a powerful linear algebra library, that uses *BLAS* and *LAPACK* libraries ³ to utilize the available performance to its maximum.
- Latest *GCC* and *Clang* compilers: This is often a neglected aspect, but it can be important to performance. As the compiler is designed to optimize the code to maximize the desired performance (if `-O3 -march=native` are used), and more recent compilers will be designed to use the more recent instruction sets of CPUs.
- *OpenMP* multi-threading with shallow copies of control systems, allowing uniform distribution of the simulation work to a very high number of threads (in use-case, we can use up to 32000 cores), without unnecessary copies.
- Strategic usage of memory allocations and references, to avoid allocations or deallocations of heap memory in hot paths.
- Smart use of search algorithms that are optimized for monotonic functions (which allow galloping search), for finding two values simultaneously (avoiding double search), or for finding the nearest point in a range (allowing for narrow cache optimized range search based on the robot's speed and position) ⁴.

An interesting expansion of this would be to incorporate GPGPU acceleration to the simulation. However, in our use cases we did not have any large enough matrix multiplication, or other compute candidates that would benefit from this. If this would be the case, then the least intrusive approach would be to use *CUDA* acceleration for the *Eigen* library. It should be noted that the simulator is designed for multi-threading in mind, as such GPGPU acceleration would conflict with the parallelization strategy, which could cause a significant slow down.

In order to understand the simulator's performance, here is a table with the percentage of CPU cycles spent for each task when simulating:

- 74.2% Robot model & ODE solver, of which:
 - 20.5% is from the model calculations and ODE solver
 - 53.7% is from trigonometric function calls
- 13.2% Neural Network prediction
- 5.2% Sliding observers
- 4.6% Kalman filter
- 1.0% Controller
- 0.4% Path tracking
- 0.2% Dynamic observer

³CPU brand specific variants such as *Intel's MKL* or *AMD's AOCL* can provide higher performance

⁴A K-D tree search algorithm is the next best candidate, if none of these methods apply.

- 1.2% Other (i.e XML loading, getters/setters, random number generator, etc...)

An additional speed up can be obtained with the robot's dynamic model. Indeed, most of the sliding angles are between $\pm 5^\circ$, and when this occurs we can use the approximation $\arctan(x) = x$ and $\sin(x) = x$ when calculating the sliding forces using the Pacejka model Eq. 2.5, which gives us the following equation:

$$F_y = DCB\alpha \quad (2.22)$$

In practice, this allows for a 47% speed up when computing the robot model (35% when comparing overall), with a 0.5% error on the lateral forces.

Overall, the simulator is capable of a 1000 : 1 ratio of simulated time to CPU time on a Ryzen 9 3900x CPU. Meaning a 1h real world wall time of simulation with 24 CPU cores in parallel, will simulate $3600 * 24 * 1000 = 86400000$ s, or more readably 2.7 years of simulation for every real world hour.

Extending beyond the simulator

The simulator was designed to be modular in nature, so the controller and the robotic model could be swapped in a relatively short amount of time.

Furthermore, for the sake of compatibility between different codes, the neural network (that will be detailed in the following section) is saved to an xml file, with csv formatting. This is so the open format can be easily loaded to *ROS* or other codes, and allows developers to quickly understand the file format, making it as portable as possible.

Chapter 3

Reinforcement learning approach to robotic control

3.1 Overview of the machine learning methods

When studying the scope of machine learning methods, three commonly used categories can be derived from the way they obtain their error signal. They are called self-supervised learning, supervised learning, and reinforcement learning.

Self-supervised learning

Self-supervised learning (also known as unsupervised learning) is a set of machine learning methods, that are able to determine patterns from a given dataset. These patterns can be used to analyze and compare future data points, or to generate artificial data points that are similar to the initial dataset. These methods are usually used for categorization, clustering of data, lossy compression (or dimensional reduction), error detection, and data generation.

Some examples include variational auto encoders [57], as they are able to compress a given data point from a data set into a more compact representation; and generative adversarial network [58], as they are able to create novel artificial images after training with existing images.

With the robotic control problems, the optimal control output is not in the set of observations, further more, generating observations would not be of much use for controlling a robotic system. As such, self-supervised learning will be ruled out as a research possibility.

Supervised learning

Supervised learning is a set of machine learning methods, that are able to train an approximate regression such as predicting house pricing or classification such as image recognition [16]. To do this, these methods require that the given input dataset is associated with the matching target values (known as labels).

Although these methods are capable of training a model to approximate the desired mapping between the observation and the optimal control output, the methods require the optimal control output to be given for each observation of a given dataset. This task is not only difficult from the perspective of time required to label the dataset, but it is also difficult from a conceptual perspective, as it is not always clear for a given observation what the optimal control output would be. Furthermore, these methods prevent useful generalization to other robots, controllers, or environments, as the labeling must be redone for every possible configuration. However, these methods could be considered in order to improve the perception of the system for our tasks through state representation learning (SRL) [59, 60].

Reinforcement learning

Reinforcement learning is a set of machine learning methods, that are able to train a policy π , which for an observation \mathfrak{s}_t at a time t , predict the action a_t : $\pi(\mathfrak{s}_t) = a_t$. This action from the policy is trained to maximize the immediate and future reward r from a user defined reward function [17]. These methods are able to outmatch expert humans at the game GO [4], and allow

for complex control policies that would be hard to define mathematically, such as finding a control policy for a robotic hand that can solve a Rubik's cube [20].

These methods are capable of finding the mapping from the observation space to the optimal control output, encoded in the policy, where the action is the optimal control output. As such, this class of methods fit the robotic control problematic. The reward can be defined in many ways. To begin however, it will be defined as the negative of the control errors, so that a decrease in the error will increase the reward that the method is trying to maximize. As such, the policy for every observation will give the control output that minimizes the current, and future control errors.

A Markov modeling for robotic control

Time difference reinforcement learning as described by Sutton[17], is based on a Markov decision process.

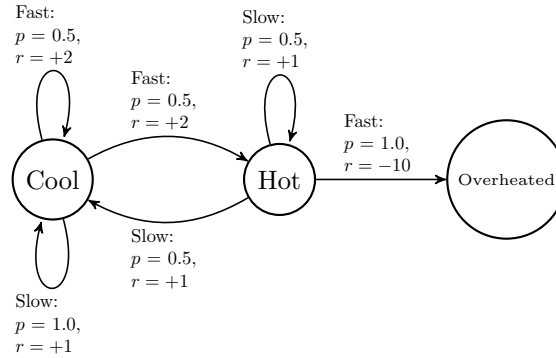


Figure 3.1: An example of a Markov decision process for a "racing car". Each state represents an environmental state and configuration (position, speed, ...), the action influences the next state, and the reward is the quality of the transition between two states.

A Markov decision process (MDP) is an extension to the Markov chain, where for a given state \mathbf{s}_t , an action a_t will change the transition probabilities to the other states in the MDP, and when a transition between two states occurs, a reward r_t is returned. By defining a policy $\pi(\mathbf{s}_t) = a_t$, the MDP is reduced down to a Markov chain. The MDP models the environment, and the policy models the agent in the environment. An example of a MDP can be seen on the figure 3.1.

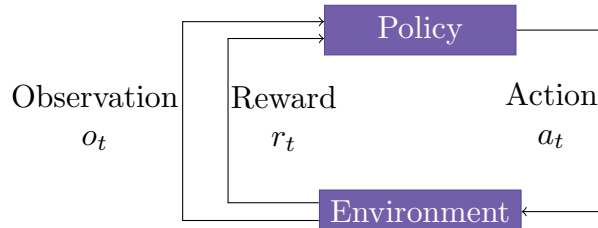


Figure 3.2: A block diagram of a control loop using reinforcement learning.

With the Markov modeling, the control loop can be reduced to a feedback loop between the environment and the policy (figure 3.2), where the goal is to minimize the cumulative control error, by taking actions from the observations.

In the case of controlling mobile robots, the observation is not a direct reflection of the state of the environment, as the sensors are not perfectly accurate. Indeed, there is information that is hard to measure such as the sliding of the robot, and the observation has to also encode the temporal information in order to determine an approximate state of the robot.

These temporal problems, linked with the observability of the robot's state in the environment, make the Markov decision process a partially observable Markov decision process (POMDP). This means the reinforcement learning methods will be impaired, and the training will be harder [17]. Most of the observation noise will be mitigated by an extended Kalman filter, meaning the noise from the sensors should be reduced substantially but not removed completely.

3.2 Time difference reinforcement learning

Time difference reinforcement learning consist of exploiting the immediate and previous rewards, in order to estimate the expected reward for a given state (i.e. Value function), or for a given state action pair (i.e. Q-value function) [17]. This allows these methods to estimate the expected reward while exploring the environment, instead of at the end of each episode.

Most deep reinforcement learning methods use time difference in order to be sample efficient and as such learn quickly. For this, a *critic* is used to determine the Value function and Q-value function, defined by a neural network. The resulting *critic* can then be used in order to determine a policy for controlling the system, in order to maximize the expected reward. However, this means that these methods are strongly dependent on the quality and capability of training said *critic*, as in practice the gradient of the *critic* is used to determine the policy function [61, 26, 62].

Value function

In reinforcement learning, the concept of value function is defined as the cumulative sum of the current reward r_t and future reward r_{t+1}, \dots, r_T of a given state \mathbf{s}_t .

The goal in reinforcement learning, it to maximize the reward over time. The value function then acts as an objective target to be optimized [17].

The value function is defined as such:

$$V(\mathbf{s}) = \mathbb{E} \left[\sum_{l=0}^{T-t-1} r_{t+l+1} | \mathbf{s}_t = \mathbf{s} \right]$$

It is the expected cumulative reward from a given state. Using reinforcement learning algorithms, a policy can be optimized that takes actions towards states that have a high cumulative reward.

It can be very costly to estimate the value function at the end of each episode, as such time difference [63] is used to update the estimation of the value function at every timestep, as opposed to the end of each episode. An episode is defined as the duration between the time the Markov decision process started in the initial state and the time it reached a termination state (for example a racing robot from starting to reaching the finishing line is an episode).

By applying the Bellman optimization (as described in the appendix, section A.1), the following equation is obtained:

$$V(\mathbf{s}) = \mathbb{E} [r_{t+1} + \gamma V(s_{t+1}) | \mathbf{s}_t = \mathbf{s}]$$

Where after a transition from \mathbf{s}_t to \mathbf{s}_{t+1} , the value function can be updated using the immediate reward r_{t+1} , and it's own estimate of itself for the future reward $V(\mathbf{s}_{t+1})$, bootstrapping it's approximation of $V(\mathbf{s}_t)$, and reducing the number of timesteps needed to converge to the exact value of $V(\mathbf{s}_t)$. The name *time difference* is derived from the equation using the future estimate of the value function as a target for the estimation at current state of the value function.

Action policy

The goal of the policy is to map the quasi-optimal control output from the environment observation. This policy must be configurable by a vector of parameters that is given from the optimizer, implying that the policy is a function approximator.

Optimizing the parameters of a function approximate is defined as policy optimization in reinforcement learning.

A function approximator is a function that is capable of closely match a target function. Many classes of function approximators exist, each with its strengths and weaknesses. The following is a short list of compatible function approximators that can be used for the task.

- Lookup table: where range of values defines a target value. This function approximator would be similar to the fuzzy logic methods, as they both output discrete values for a range of input values (such as tabular-TD [17]).
- Polynomial: A natural extension of the linear approximator, and can be seen as a Taylor-Young approximator of a function [17].

- Neural networks: Currently one of the most popular function approximator in machine learning, it is defined as a universal function approximator [10]. This means with two hidden layers of enough neurons, a neural network can approximate any function [17].

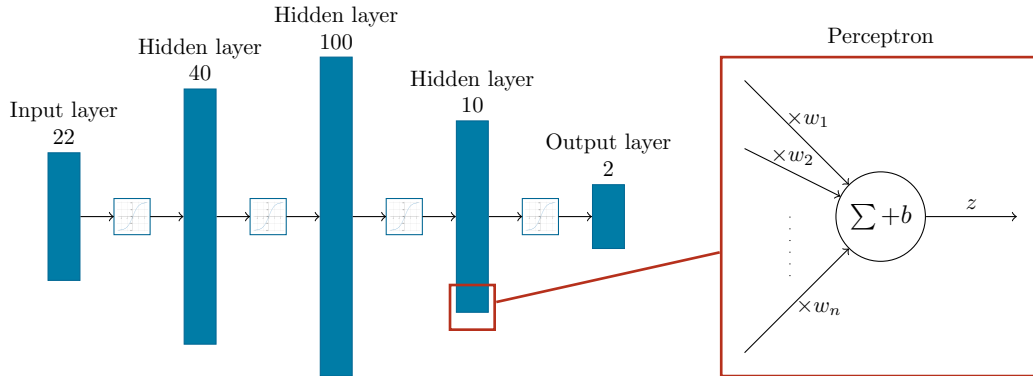


Figure 3.3: An example of a policy model.

A neural network is a sequence of matrix transformations with a bias vector, separated between activation functions. Each neuron is taking the output of the neurons of the previous layer, multiplies it by a weight w , sum those weighted outputs together, and add a bias value. An example of this can be seen on the figure 3.3. The more hidden layers and neurons in the neural network, the higher the capacity the neural network has to match the desired function, as shown in [10]. However, this also means that a neural network with a higher number of layers and neurons than is required, can overfit to the training data, which will not generalize well.

A key aspects in defining a neural network lies in the determination of the exact layer sizes, the number of layers, and the shapes for activation functions. It indeed relies on different characteristics and complexity of the desired target function to estimate. As a result, despite some theoretical works exist on the subject [64], it remains a non trivial problem. As such, a grid search (an exhaustive generation of search parameters, generated from a list of parameters) is run in order to test multiple architectures of neural networks, as a neural network that is too small would not be able to encode the desired action policy, but a neural network that is too big would be harder for certain optimizers to optimize due to the curse of dimensionality.

Due to the temporal independence nature of the neural network with respect to the input, the neural network will not have any persistence between two observations. As such if temporal information is required in order to predict the desired output, the information either needs to be integrated into the input vector, or the neural network needs to have a recurrent method such as an LSTM [65].

Existing methods

In order to implement these concepts with neural networks and deep learning, there exists a large body of commonly used methods. The first of which popularized to play Atari games in the journal Nature is the Deep Q network (DQN) from [19], uses a neural network to predict the Q-value (A.1) for each discrete action from the state vector. In our case we need multiple continuous outputs, as such the extension of DQN, Deep deterministic policy gradient (DDPG) from [62] is a good candidate we can consider. Altering DDPG by moving the actor and the critics from a sequential configuration to a parallel configuration, and by training using the value function rather than the Q-value, leads us to Advantage actor critic (A2C) from [26], Proximal policy optimization (PPO) from [61], and Soft actor critic (SAC) from [66]. These are methods that have proven to be good methods in the field of reinforcement learning, and are often used as baseline methods.

These methods are tested using the RL library Stable-Baselines [67], on controlling the steering angle of a simulated kinematic mobile robot of $500kg$ with an action delay of $0.5s$ at a control rate of $10hz$, trained over canonical trajectories such as a straight *line*, *sine* curve, a *parabola*, and two *splines*. From this, the results in Table 3.1 are obtained. It seems that these methods struggle to converge on some trajectories, and overall have very low performance compared to the existing

Romea controller. What is paradoxical, is that these RL methods have shown promise in closely related environment, such as *CarRacing-v0* from Open-AI's Gym [68].

As such, this implies that there is a fundamental difference that exists from our use case, that prevents the correct convergence of these methods.

trajectory	Existing controller	SAC	PPO	DDPG	A2C
line	48.20 (± 2.45)	60.99 (± 22.21)	115.11 (± 57.10)	158.14 (± 4.09)	57.52 (± 15.12)
sine	151.70 (± 2.83)	1140.52 (± 3004.65)	2403.93 (± 2824.32)	309.18 (± 5.79)	280.51 (± 175.61)
parabola	125.29 (± 4.02)	191.48 (± 2.43)	389.65 (± 6.34)	417.99 (± 6.23)	208.80 (± 26.57)
spline1	142.04 (± 3.45)	508.24 (± 152.24)	2864.41 (± 14.31)	272.20 (± 5.27)	542.85 (± 4.47)
spline2	164.94 (± 4.45)	6563.63 (± 3310.81)	3169.80 (± 23.61)	258.85 (± 26.39)	437.22 (± 736.97)

Table 3.1: The values of the integral of the lateral error over time (in *m.s*) for every trajectory with Time difference reinforcement learning methods.

Limitations

At the time of writing, the most studied class of reinforcement learning methods is model free temporal difference reinforcement learning [4, 20, 67], which updates its actor at each iteration of the control loop, and does not require an existing system model in order to converge. However, these methods cannot be used in here, due to the following triad of issues:

Observations at high frequency control

In many control systems, a control loop frequency higher than 1Hz is not only expected, but necessary. Unfortunately, the Markov decision process that temporal difference reinforcement learning is based on, requires significant inter states differences in order to converge. And as such, a high control loop frequency might cause observation issues for the Markov decision process.

Action delays and inertia

Real world systems that are large enough, are subject to strong inertia and action delays. These are caused by the square-cube law, which implies the volume and weight of an object scales to the cubed of the length. This causes inertial to no longer be negligible, which mean these systems have a significant delay between a requested action, and the action taking place. This delay on the action causes a dissociation between the states, rewards, and actions. Indeed, an action taken by the reinforcement learning method at time T , will affect the system at a time $T + n$, which means that the observation of the effect of the action taken at time T will only be observed at a time $T + n$. This effects are even stronger when considering the potential "pure" signal delay between the command being computed and when the command is received by the actuator. And as such, this might cause convergence issues when the delay n is large enough, as the method will encounter a credit assignment problem, as described in [69].

Noise cancellation of second order filters

Most mechanical systems have significant physical constraints. Indeed, an actuator has a limited action range and speed, but also a limited acceleration and jerk. These limitations mean that an actuator system can be modeled by a low pass second order filter, where a desired action controls an acceleration. One of the unfortunate side effect of a low pass second order filter on the actuators, is that it will cause a filtering of any high frequency. This means that noise applied to a control signal will be nullified if the control frequency is high enough. And since reinforcement learning methods use noise to explore the environment, it suggests that the exploration of the reinforcement

learning method will be nullified. This has already been described previously, and solutions have been proposed [70].

3.3 Transition to episodic

Due to these limitations, popular methods such as DDPG, A2C, and PPO cannot be used, as they depend on time difference reinforcement learning. As a result, a different class of reinforcement learning methods must be considered.

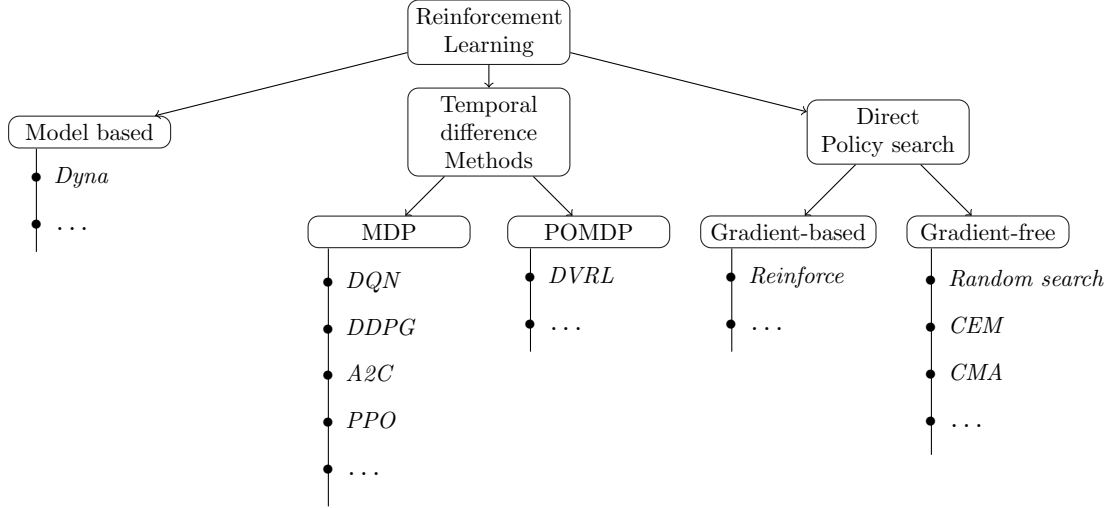


Figure 3.4: A hierarchical diagram of some of the types of reinforcement learning methods.

Figure 3.4 depicts a classification of commonly used types of deep reinforcement learning methods. The ones discussed up until now were the *Temporal difference methods*, and more specifically the ones based on a *Markov Decision Process* (MDP). As shown previously, these methods are not capable of converging correctly on the desired system, due to their intrinsic features. As such, the category of *Temporal difference methods* must be unfortunately excluded if we wish to conserve a high control frequency.

An alternate category would be *Model based* methods, which seem promising as they use a model of the system in order to generate an action policy (e.g. [4]), however they generally use a *Markov Decision Process* (MDP) as a modeling tool, in order to apply a planning strategy for their policy. This means they will also suffer from the same intrinsic features of the previously considered systems, and as such will also struggle to converging correctly at a high control frequency.

The final category described is the *Direct policy search*, which consists of using an optimizer to directly tune the action policy from an *objective function*. This means that these methods do not require a *Markov Decision Process* (MDP) as a modeling method, and do not require an estimation of the value function in order to converge. This isolates these methods from the previously described issues relating to the intrinsic features of our system.

Within the *Direct policy search* methods, we can describe two subcategories: *Gradient-base* methods and *Gradient-free* methods. The *Gradient-base* methods depend on a gradient between the output of the action policy and the objective function. In our use case, assuming the objective function is proportional to the error, the following gradient is derived:

$$\frac{\partial obj}{\partial u} \propto \frac{\partial e}{\partial u} = \frac{\partial e}{\partial X} \frac{\partial X}{\partial y} \frac{\partial y}{\partial u}$$

obj is the objective function, u the control vector, y the measurement vector, X the state vector, e the error vector. From this, $\frac{\partial e}{\partial X}$ denotes the rate of change of the tracking error, $\frac{\partial X}{\partial y}$ denotes the rate of change of the state estimator, and $\frac{\partial y}{\partial u}$ denotes the rate of change of the robot and sensor's model. This shows that the gradient from the objective function to the control output from the action policy, is quite complex, non-linear, and noisy due in part to sensor accuracy and model accuracy. As such, it would be sub-optimal for these methods.

The *Gradient-free Direct policy search* methods category on the other hand seems like a good candidate, as these previous works [71] seem to suggest it has comparable performance to *Temporal difference methods*. As such, it is the category of methods that will be used in the following.

3.4 Gradient-free Direct policy search

An alternative to time difference

In some tasks, a Gradient-free Direct policy search using the optimizer called *Natural Evolutionary Strategy* (NES) from [72] was shown to have equivalent performance when compared to existing time difference reinforcement learning methods [71]. As such, they are a promising approach for the desired task.

Gradient-free Direct policy search methods differ from the classical time difference reinforcement learning methods in the following ways:

- No Reward: Instead an objective function is used, and as such the optimization target $G(s)$ is directly written by the developer, and not deduced from integrating the reward over time.
- Episodic Training: i.e updating the neural network at the end of each episode, where an episode is delimited from a starting state to a terminal state. As opposed to updating the neural network at each timestep.
- No critic: In order to obtain a gradient for the action policy, a time difference reinforcement learning method needs a critic that needs to learn the Q-value or value function. However, since a gradient-free method is used, no critic is needed, which reduces the search space, and helps to distance these methods from a Markov modeling.
- Moving from gradient descent (e.g PPO, A2C, DDPG, ...) to a black box optimizer due to the gradient not being available here with a lack of a critic. As such, the optimizer changes the weights & biases of the neural network directly.

Moving from reward to objective function

Usually in time difference reinforcement learning, the reward is used as a target for the optimization process through gradient descent. This reward is defined as the value that is maximized when a desired behavior has been taken. It is returned at each timestep between two observed states.

However, when using gradient-free direct policy search, an objective function needs to be used in order to be compatible with black box optimizers for direct policy search. An objective function in our case is obtained at the end of an episode, and is minimized when a desired behavior has been taken.

The objective function takes the set of all the observed states during an episode, and then returns a scalar value that express the performance of a given episode. This allows for arbitrary integration to take place in the objective function.

More formally, it takes a set of states of size k taken over a time span T defined as $S \in \mathbb{R}^{n \times k}$ where, $s \in \mathbb{R}^n$, $k \in \mathbb{N}$, and $s_t \in S$ for any $0 \leq t \leq T$, and returns an objective function value in \mathbb{R} . This leads to our definition for our objective function as: $obj : \mathbb{R}^{n \times k} \rightarrow \mathbb{R}$.

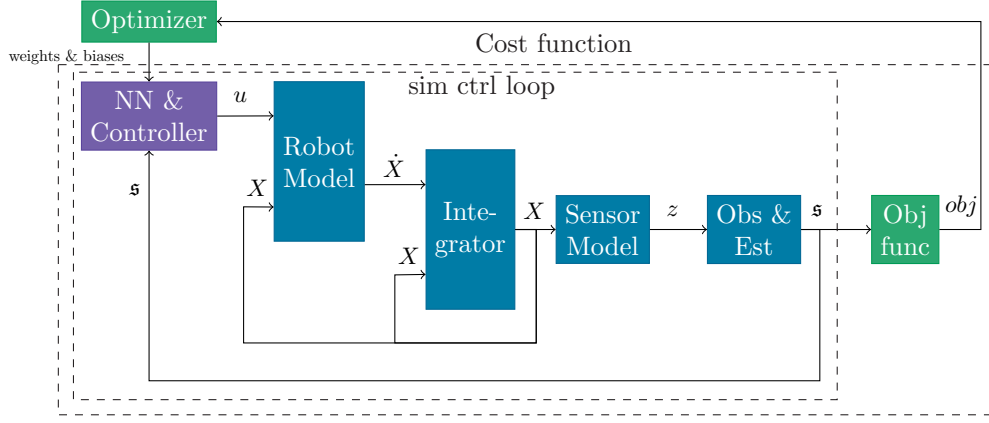


Figure 3.5: The full training loop with the objective function and optimizer.

The figure 3.5 denotes how each element of the control loop interacts in the simulation (through the RK4 integrator), and with the optimizer. The *sim ctrl loop* denotes the inner control loop that is run over the trajectory, and the *Cost function* denotes the system that the black box optimizer is optimizing guided using the objective function that the *Cost function* returns at the end of each episode.

Optimizers for episodic reinforcement learning

A black box optimizer must be used for gradient-free direct policy search. Due to lack of time, restart strategies associated with optimization method [73, 74] have not been explored, however these paths could lead to higher performance. For the direct policy search a first natural and trivial candidate optimizer to test is a *Basic random search* algorithm.

Basic random search

The *Basic Random Search* (BRS) method (from [75]) is based on the following idea: take a vector of independent and identically distributed variables of the same size as the search space. Then add and subtract this vector to the mean value, in order to create two candidates for the population. Once both of these candidates are evaluated, update the mean proportionally to the evaluated return of the objective function, so that the mean tends towards the minimum of the search space. In some tasks, this method has shown surprisingly good results despite the simplistic nature of the algorithm [75].

Unfortunately, this method yields poor performance due to the constant variance (or stepsize) of the sampled vector, causing *Basic random search* to find solutions around the local minimum, but being unable to reduce its stepsize in order to find solutions within the local minimum.

Details of this algorithm and a comparison with the other optimizers can be seen in appendix A.2, from this the performance of BRS seems very low compared to the other methods, and as such is discounted.

CEM

A more complex method we can consider is the *Cross-Entropy Method* (CEM) from [76]. It consists of sampling candidates of a population \mathcal{P} from a mean μ and a standard deviation σ using a normal distribution:

$$\mathcal{P} \sim \mathcal{N}(\mu, \sigma^2)$$

Once the population \mathcal{P} is evaluated, an elite selection (usually $\frac{1}{5}$ th of the population size) is selected as the basis for the mean μ and a standard deviation σ of the next generation, which allows for a step by step iteration towards the nearest local minimum.

However, this method returned a sub-optimal performance when used in training, due to its fast convergence towards the nearest local minimum, when potentially better minimums might exist nearby.

Details of this algorithm and a comparison with the other optimizers can be seen in appendix A.2. From this, the performance of CEM is consistent and better than BRS, but substantially lower than the next method, CMA-ES.

CMA-ES

In the world of stochastic optimization, few strategies are able to reach the widespread use of CMA-ES; in part because of its capacity to optimize high dimensional problem spaces, and its capacity of optimizing highly modal problems (meaning problems with many local optimums) [77].

CMA-ES (Covariance Matrix Adaptation - Evolution Strategy) from [78] is an evolutionary strategy used for stochastic optimization of a problem space over a given objective function. In order to do this, CMA-ES calculates an estimate of the covariance matrix over each dimension of the problem space for the target objective function.

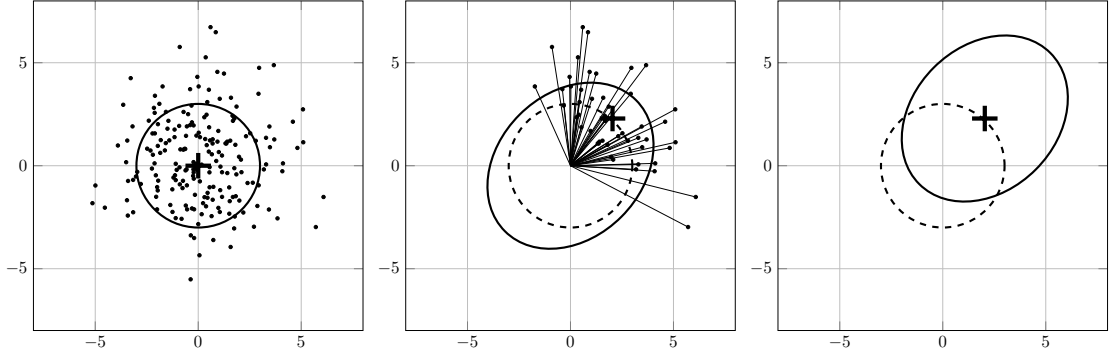


Figure 3.6: The CMA Evolution strategy, each point is a sampled population, the cross is the mean, the full circle is the covariance, and the dotted circle is the old covariance. Left: the initial population sampling from the mean and the covariance. Middle: the elitist selection after the evaluation, updated mean and covariance in the direction of the local minimum. Right: New covariance and mean for sampling the next generation

$$\mathcal{P} \sim \mathcal{N}(\mu, \sigma^2 C)$$

The population \mathcal{P} , generated by the CMA-ES method, is sampled from multivariate Gaussian distribution with a mean μ , a global variance σ , and the covariance of the search space C . At each iteration, the population is evaluated, and from the population with the lowest objective function score, the covariance, mean, and global variance are updated toward the objective function's local minimum.

This allows the CMA-ES method to find which parameters are covariant between each other with respect to the objective function, and avoids sampling solutions that do not converge towards the local minimum of objective function. In theory, it accelerates the convergence speed of the method.

Sampling over the covariance matrix can also be seen as projecting the problem space through the accuracy matrix (inverse of the covariance matrix), allowing for a random search in a normal Gaussian space.

$$\mathcal{P} \sim \mu + \sigma C^{\frac{1}{2}} \mathcal{N}(0, I) \iff C^{-\frac{1}{2}} \mathcal{P} \sim C^{-\frac{1}{2}} \mu + \sigma \mathcal{N}(0, I)$$

When tested over the environment, this method shows the highest performance out of the tested methods (see appendix A.2). The CMA-ES method having been confirmed as the best solution from the tested methods, we then proceeded to compare some of the CMA-ES variants (see appendix A.3), in order to determine an ideal method for the training, in order to obtain solutions that have the desired performance, without having a compute cost that is too high (computing cost versus solution performance).

Due to neural network's parameters being roughly independent, we can use the separable variant of CMA-ES, sep-aCMA-ES [79] (as shown and explained in appendix A.3). As such, sep-aCMA-ES will be used in order to optimize our neural network for controlling a mobile robot.

3.5 CMA-ES based training in simulation

Due to the nature of CMA-ES, training is not practical in real world conditions due to its parallel nature of comparing many candidates in similar conditions.

As such this approach has two phases. The first one is a training phase in simulation, using CMA-ES in order to determine an optimal neural network. And during a second phase, the trained neural network is used by itself without CMA-ES and in real time, in order to control the mobile robot from the input state.

During the training phase, the neural network given as a starting for CMA-ES is defined randomly using a Xavier initialization [80]. This allows for an initial randomized output from the neural network, rather than having a neural network set to 0. This allows for the exploration rate σ to be considerably reduced, and allows the CMA-ES method to spend more compute time optimizing the parameters of the neural network. In practice, this improved the training of the neural network, allowing it to reach lower values in the objective function.

Furthermore, consistency issues with training and qualities of local optimal can be observed. In practice this meant that any two training generated very different neural networks, with varying performance. For the first aspect of training consistency, the use of regularization of the neural network can be used, in order to reduce overfitting and find more general solutions to the task. A natural fit for this would be *batch normalization*, as it allows for regularization without applying the dropout method which can hinder the training process. However, the use of batches when applied to prediction is not practical, as in the context of reinforcement learning the next observation is directly dependent on the current action that needs to be taken, which means that the observation cannot be stacked into a batch to be computed simultaneously. As such, we used a similar methodology as described in [71], which consists of generating an example batch for normalization¹, that is then used throughout the training with the CMA-ES method. This batch is generated in order to describe a close approximation of the distribution of each input parameter, which can be done using an existing controller that is tuned with constant gains.

This setup of CMA-ES and the neural network is then trained over a set of canonical trajectories and trajectories derived, in order to reach a full dataset of varying conditions. Where the

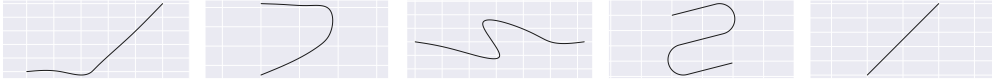


Figure 3.7: A representation of the trajectories used in training.

trajectories are depicted in figure 3.7 and detailed in section A.12.

These trajectories are tested at multiple speeds, with varying grip conditions, multiple times (in this work 5 times) in order to increase the consistency of the result by using the *Law of large numbers*. Overall, each neural network candidate is tested approximately 1000 times (5 trajectories with 2 scaling factors, 5 times, with 5 grip conditions, at 4 different speeds), before the final cost value is determined, and returned to CMA-ES, in order to quantify the given candidate neural network. This is then repeated for the entire population of neural networks P that CMA-ES sampled (in our case, we chose a population size of $N_{\text{pop}} = 32$, from [78] where $N_{\text{pop}} = 4 + \lfloor 3 \log(N) \rfloor$, with $N = 15000$ being the number of NN parameters that CMA-ES tunes). CMA-ES is then used in the simulator previously described.

3.6 Neural network architecture

The neural network chosen must be well constructed. Indeed, if it is too small it might not be able to encode the desired transformation between the input state and the desired control output. However, if it is too big, then the CMA-ES optimizer might not be able to properly converge due to the curse of dimensionality, and the large search space. As such, a compromise between training performance and neural network capacity must be reached.

The exact layer size, number of layers, and the kind of activation function are all meta parameters that are difficult to know in advance for a given task. As such, a grid search is run in order to test multiple architectures of neural networks, along with some trial and error.

¹The batch is generated of random samples using the reservoir sampling algorithm.

First, a neural network of a single hidden layer is defined and tested, as it is a universal approximator [10]. The size of the hidden layer is then incrementally increased, until the CMA-ES method's training performance (measured through the objective function) starts to reach a plateau. This means that we have reached the maximum number of parameters of the neural network, while still being trainable by CMA-ES.

Then, the neurons on a hidden layer are split in such a way to increase the depth of the neural network (increasing its non-linearity), while preserving its number of parameters. The performance is then compared with the previous iteration, and is preserved if the performance is better. This is done until the number of layers is set.

Finally, the neurons are distributed across the layers, in such a way as to improve the performance. Generally, wide layers can interpolate more intermediate values when compared to small layers.

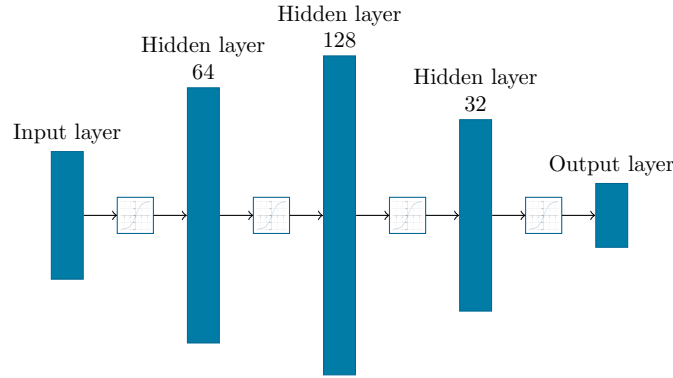


Figure 3.8: The neural network architecture used in the following works.

From this, a fully connected neural network of size 64,128,32 reached a good balance between a low number of parameters (approx 15000), and good performance in our use case, and as such is the neural network used in the rest of the work presented. Of course, this values are very dependent on the problem, and should be rechecked periodically as the task or the inputs might require more or less parameters in order to optimally approximate using CMA-ES. In this work, over 20 unique architectures were tested, in order to reach the final neural network architecture, each one taking about a day to train and validate.

3.7 Reinforcement learning strategy selection

In this chapter we have discussed several methods to apply machine learning for off-road mobile robot trajectory tracking. We saw the limitation of different strategies (such as time difference reinforcement learning, supervised learning, and unsupervised learning). From which we propose the following solution: Applying episodic reinforcement learning using policy iteration, over a neural network with 3 hidden layers and tanh activation functions, that is optimized using sep-aCMA-ES with a population of 32 for 20000 iterations. The analysis of the neural network will be assisted by a feature importance strategy, as described in section A.4

Based on this configuration, we will now investigate several strategies for the proposed training method to act on the autonomous driving system. In particular a classical End-to-end approach called *NN controller*, a corrective approach called *Delta NN ctrl*, and a parameter tuning method called *NN gain tuner*.

Chapter 4

Applying reinforcement learning for robotic steer control

4.1 Direct steer control using Reinforcement learning [*NN controller*]

An ideal first approach to implementing reinforcement learning to robotic control, would be the canonical approach in reinforcement learning, which consist in controlling directly the robot with a neural network using the current state of the system and the reference trajectory as inputs. In this chapter, the steering will be controlled in order to compare with our reference controller for mobile robot steering control, and the speed is set to a constant value in order to simplify the analysis initially.

Experimental setup

Control loop setup

In this part, the neural network directly controls the steering of the mobile robot's system, as shown in the figure 4.1 where the neural network takes the errors, curvature, and speed of reference, then returns the predicted steering control output.

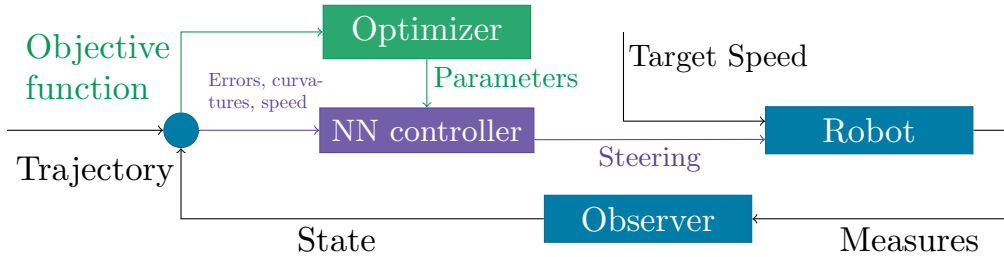


Figure 4.1: Overview of the proposed method.

The neural network is trained in a simulation using the CMA-ES optimizer, depicted as the *Optimizer* in the figure 4.1, which takes the objective function value and returns the parameters. The neural network takes as input the same information as an existing steering controller, which is the lateral error, angular error, curvature, future curvature (20 sampled points over a 5s horizon of prediction, chosen in order to approximate the useful future curvature within 3 times the effective delay, while avoiding sub-sampling problem and without increasing the number of inputs by too much), speed, and the robot's steering state. And as control output the steering angle, with the speed control output being defined as constant (similarly to [24, 21, 25, 27, 28]). An Extended Kalman Filter (EKF) [53] is used as the *Observer* in order to filter the noise from the robot's sensors, and improve the accuracy of the tracking. The *Robot* block is the dynamic model of the robot in section 2.3, using a Runge-Kutta (RK4) integrator.

Objective function

The objective function is defined as a composition of different targets. Indeed keeping a minimal distance to the trajectory is not sufficient as an optimization target, as it does not prevent from oscillations when the lateral error is low enough. As the function needs to return a scalar value from a set of sampled state vectors (as shown in section 3.4), an integration must take place. As such, an absolute value¹ discrete integration over the curvilinear abscissa is done, in order to avoid any side effects due to speed modulation. The result of the integration is then normalized over the length of the trajectory in order to keep the objective function values consistent between each trajectory.

The first component defined obj_{err} , describes the penalty over the lateral error. Where N is the number of samples recorded over the trajectory, s_N is the length of the trajectory, and the remaining notation is consistent with convention defined in section 2.2 and in section 2.3:

$$obj_{err} = \frac{1}{s_N} \sum_{i=0}^N |k_{yi} y_i| \Delta s \quad [m] \quad (4.1)$$

Where k_{yi} is a dynamic objective function parameter, that will change the lateral error penalty if the lateral error exceeds a given limit y_{lim} :

$$k_{yi} = \begin{cases} k_{y \text{ low}} & \text{if } |y_i| \leq y_{lim} \\ k_{y \text{ high}} & \text{else} \end{cases}$$

And where Δs denotes the difference of the curvilinear abscissa over the time difference Δt :

$$\Delta s = v \cos(\tilde{\theta}) \Delta t$$

The second part of the objective function is the penalty over the steering angle, as the robot's steering should match the steering needed to follow the curvature of the trajectory:

$$obj_{steer} = \frac{1}{s_N} \sum_{i=0}^N |Lc(s) - \tan(\delta_{Fi})| \Delta s \quad [] \quad (4.2)$$

The final composition of the objective function can now be defined, with k_{steer} as the proportional gain between the steering angular error penalty and the lateral error penalty (measured in meters). It is important to note that lateral error and angular error are not opposing targets, as they will both tend to be positively correlated and moving in tandem with each other. This means that they can both be added in a linear fashion, without any unusual side effect from the optimization (i.e it will not over-optimize one at the expense of the other).

As such the objective function obj_1 is defined as:

$$obj_1 = obj_{err} + k_{steer} obj_{steer} \quad [m] \quad (4.3)$$

Or more completely:

$$obj_1 = \frac{1}{s_N} \sum_{i=0}^N [|k_{yi} y_i| + k_{steer} |Lc(s) - \tan(\delta_{Fi})|] \Delta s$$

In this chapter, the objective function for training is defined with an allowed error corridor of $y_{lim} = 0.20m$. It was found experimentally through trial and error that a $k_{steer} = 3^2$, $k_{y \text{ low}} = 1$, and $k_{y \text{ high}} = 10$ returned a trained model with the ideal performance.

Surface error

The metric used for the analysis of the results is the surface error:

$$A_{error} = \sum_{i=0}^N \left| v_i \cos(\tilde{\theta}_i) \left(y_i + \frac{v_i \sin(\tilde{\theta}_i) \Delta t}{2} \right) \right| \Delta t \quad [m^2] \quad (4.4)$$

¹ An absolute value of the errors was chosen, as the square of the value yielded sub-optimal results when tested.

² k_{steer} was determined by increasing it slowly from zero, until a stable behavior was obtained.

This is done, in order to validate and compare the performance of the different tested methods and parameters, without resorting to the objective function. Indeed, when a reinforcement learning agent trains to optimize a function, it is possible that the said agent might exploit the objective function in order to minimize it, without achieving the desired behavior. As such, using a different metric to measure performance allows for minimal bias when comparing the methods.

Training details

The neural network is trained over 5 unique trajectories (*estoril5*, *estoril7*, *estoril910*, *line*, and *spline5* as shown in A.12) twice with two varying scaling factors of 1 and 2 (this is done so longer trajectories are also tested with lower curvatures), at speeds of 1.0, 2.0, 3.0, and 4.0 $m.s^{-1}$ (due to training difficulties, the speed is limited at 4.0 $m.s^{-1}$), with varying grip conditions (cornering stiffness ranging from 7000 to 30000). Properties of the robuFAST experimental mobile platform are used as parameters: a wheelbase of 1.2m, 430kg of weight, a max steering angle of 15°, and a max acceleration 0.5 $m.s^{-2}$. The training took approximately 12h over 24 cores of a high end CPU, with a decreasing objective value that plateaued which seems to indicate a successful training of the system to the possible local minimum.

Simulated results

The trained method along with the baselines controllers were tested in the simulation with the same speeds, trajectories, and grip conditions as used in the training of the trained method. These tests were run 100 times each in order to get a consistent mean and variation for each method.

In order to compare the performances of the neural network approach with deterministic methods a baseline using constant gains for the control methods needs to be defined. The forthcoming results will be compared to the controller defined in equation 2.13. As has been mentioned previously, the *Romea* control parameters relies, among others, on the robots speed. As a result, to highlight the efficiency of proposed neural network approaches, the gain of the deterministic approach has configurations with respect to the desired speed. The following table shows the selected parameters for a conering stiffness of 16000 $N.rad^{-1}$, with centimetric position accuracy. These parameters are defined up to 4 $m.s^{-1}$, as the methods trained in the nexts sections were not able to converge with a speed higher than 4 $m.s^{-1}$. The second deterministic control approach, acting also as a second

	1.0	2.0	3.0	4.0
$k_p [m^{-2}]$	1.0	0.7	0.4	0.4
$k_d [m^{-1}]$	0.25	0.1225	0.01	0.01
$H [s]$	0.5	0.5	0.5	0.5

Table 4.1: Control parameters used with *Romea*.

reference, and defined in expression 2.14 relies on some parameters, also related to the robot speed. This controller *EBSF* was tuned with the following gains for each speed: Where the horizon of

	1.0	2.0	3.0	4.0
$k_p [m^{-2}]$	24.0	24.0	24.0	24.0
$k_d []$	120.0	120.0	150.0	150.0
$k_{dd} [m^2]$	140.0	210.0	300.0	300.0
$H [s]$	2.0	1.0	0.66	0.5

Table 4.2: Control parameters used with *EBSF*.

the *EBSF* control law is defined over a distance of 2m, and as such is dynamic with respect to the speed, as shown in the table above.

Quantitative Analysis

The trained neural network has then been tested through simulation on the same trajectory and speeds used during the training, and compared with deterministic approach. A first set of simulated runs was computed, from this the table 4.3 was obtained (where the mean and standard deviation

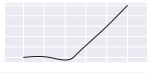



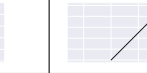
						
		estoril5	estoril7	estoril910	spline5	line
$1m.s^{-1}$	Romea	1.18 (± 0.35)	1.40 (± 0.63)	29.19 (± 0.75)	2.37 (± 1.27)	1.09 (± 0.04)
	EBSF	1.24 (± 0.37)	1.46 (± 0.70)	36.80 (± 0.65)	2.64 (± 1.16)	1.10 (± 0.04)
	<i>NN controller</i>	<u>1.43</u> (± 0.31)	<u>2.42</u> (± 0.60)	<u>21.63</u> (± 0.82)	<u>3.94</u> (± 1.08)	<u>1.29</u> (± 0.05)
$2m.s^{-1}$	Romea	1.54 (± 0.50)	2.65 (± 1.26)	33.42 (± 0.79)	4.75 (± 1.56)	1.12 (± 0.06)
	EBSF	1.55 (± 0.50)	2.31 (± 1.34)	39.75 (± 0.84)	5.76 (± 1.92)	1.11 (± 0.06)
	<i>NN controller</i>	<u>2.10</u> (± 0.33)	<u>3.02</u> (± 1.14)	<u>18.30</u> (± 0.74)	<u>5.33</u> (± 1.22)	<u>1.64</u> (± 0.06)
$3m.s^{-1}$	Romea	3.13 (± 1.07)	5.96 (± 1.50)	53.41 (± 1.40)	9.59 (± 1.27)	<u>1.21</u> (± 0.10)
	EBSF	2.09 (± 0.70)	3.54 (± 1.35)	47.21 (± 1.18)	14.88 (± 1.91)	1.16 (± 0.07)
	<i>NN controller</i>	<u>3.19</u> (± 0.52)	3.55 (± 1.20)	<u>20.59</u> (± 1.25)	<u>5.78</u> (± 1.31)	<u>2.49</u> (± 0.11)
$4m.s^{-1}$	Romea	4.31 (± 1.49)	7.90 (± 1.49)	55.82 (± 2.54)	11.29 (± 1.48)	1.28 (± 0.17)
	EBSF	3.59 (± 0.92)	4.96 (± 1.02)	51.97 (± 2.33)	92.30 (± 176.29)	2.15 (± 0.25)
	<i>NN controller</i>	<u>6.60</u> (± 1.42)	<u>7.90</u> (± 1.97)	<u>28.53</u> (± 3.25)	<u>8.80</u> (± 2.71)	<u>4.87</u> (± 0.85)

Table 4.3: Surface error in $[m^2]$ of each method at all the speeds and trajectories used during training, with an **initial error of 0m**.

over 100 runs is given). It describes the Surface error from the equation (4.4), for each method at all the speeds and trajectories used during training, with an initial error of 0m. The underlined and bold values mean that the result is significant and has a p-value³ below 10^{-3} , determined using the *Welch-t test* [81].

Overall, the average surface error for the *NN controller* is noticeably lower than reference methods at $6.69m^2$, where as the surface error for Romea was $8.39m^2$ (a 17.0% reduction), and the surface error for EBSF was $10.0m^2$ (a 33.1% reduction). From this table, the *NN controller* was able to match or exceed the performance of the existing controllers in some cases. Notably the *estoril910* and *spline5* trajectories, as they have successive corners with high curvatures which means the *NN controller* is able to learn the specific dynamics of the system and compensate accordingly, where as the existing controllers must be stable for any robot and trajectory, implying they must be sub-optimal when compared to a method that has been tuned for a given system. However, it is clear that the *NN controller* is not ideal, as it has a significant decrease in performance in the straighter trajectories such as *estoril5*, *estoril7*, and the canonical straight *line*.

From this, it seems clear that the *NN controller* is sub-optimal as it is not capable to match existing controllers with simple trajectories, and is exploiting the dynamic effects of the simulations in order to reach comparable performance when compared to the existing controllers, which can be seen as overfitting if view in the context of *supervised learning*.

Nonetheless when adding an initial lateral error of 1m at the start of the trajectory, the *NN controller* is able to reach impressive performance, as seen in table 4.4. This is due to the fact, that settling times defined by the gains chosen in the tables 4.1 and 4.2 for the deterministic control laws are set to a low values at high speed (in order to reduce oscillating behavior), which reduces the initial convergence distance. On the contrary, the NN approach allows for faster reactions, as there is no predefined nor fixed settling distance. This highlights the importance of potentially changing the control properties in an online fashion, and not only depending on the robot's target velocity.

Overall, the average surface error for the *NN controller* is noticeably lower than reference methods at $8.83m^2$, where as the surface error for Romea was $14.2m^2$ (a 38.1% reduction), and the surface error for EBSF was $15.1m^2$ (a 41.5% reduction). From this table, the *NN controller* was able to match or significantly exceed the performance of the existing controllers in every case. Notably again in the *estoril910* and *spline5* trajectories. This change in performance is not obvious from an quantitative view, as it is due to the existing controllers not being able to change their reactivity when the lateral error is large enough and/or the speed is low enough (e.g the initial

³The p-value is defined as the probability that the values obtained from the methods could be derived from the same distribution, and as such cannot be not clearly defined as significant. The lower the probability, the more likely the results are significant.

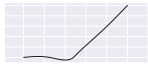



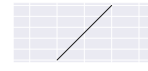
						
		estoril5	estoril7	estoril910	spline5	line
$1m.s^{-1}$	Romea	4.42 (± 0.36)	4.66 (± 0.61)	32.44 (± 0.76)	5.61 (± 1.25)	4.35 (± 0.08)
	EBSF	6.19 (± 0.37)	6.39 (± 0.69)	41.76 (± 0.65)	7.58 (± 1.13)	6.04 (± 0.06)
	NN controller	3.59 (± 0.31)	4.55 (± 0.59)	23.75 (± 0.81)	6.07 (± 1.06)	3.42 (± 0.10)
$2m.s^{-1}$	Romea	5.92 (± 0.49)	7.07 (± 1.25)	37.81 (± 0.79)	9.03 (± 1.55)	5.52 (± 0.09)
	EBSF	6.32 (± 0.49)	7.07 (± 1.32)	44.48 (± 0.83)	10.46 (± 1.92)	5.88 (± 0.09)
	NN controller	4.30 (± 0.32)	5.12 (± 1.14)	20.44 (± 0.72)	7.44 (± 1.20)	3.67 (± 0.06)
$3m.s^{-1}$	Romea	11.19 (± 1.11)	13.81 (± 1.53)	61.53 (± 1.40)	16.54 (± 1.27)	9.21 (± 0.14)
	EBSF	7.75 (± 0.72)	9.15 (± 1.33)	52.84 (± 1.16)	20.26 (± 1.87)	6.80 (± 0.11)
	NN controller	5.44 (± 0.50)	5.81 (± 1.25)	22.64 (± 1.19)	7.80 (± 1.27)	4.59 (± 0.13)
$4m.s^{-1}$	Romea	12.28 (± 1.56)	15.62 (± 1.52)	63.83 (± 2.54)	18.37 (± 1.43)	9.15 (± 0.17)
	EBSF	9.18 (± 0.96)	10.48 (± 0.98)	57.62 (± 2.46)	91.77 (± 165.52)	7.69 (± 0.23)
	NN controller	8.88 (± 1.41)	9.94 (± 1.78)	31.63 (± 5.14)	10.47 (± 2.59)	7.19 (± 0.92)

Table 4.4: Surface error in $[m^2]$ of each method at all the speeds and trajectories used during training, with an **initial error of 1m**.

section of a trajectory) due to their control parameters not adapting to the robot's state and environment.⁴

Qualitative Analysis

In order to avoid overloading the figures and due to the performance of the EBSF method, they are omitted from the qualitative analysis. When focusing with a qualitative analysis over the *spline5* trajectory, at $2m.s^{-1}$ with 1m of initial error, the following results are obtained.

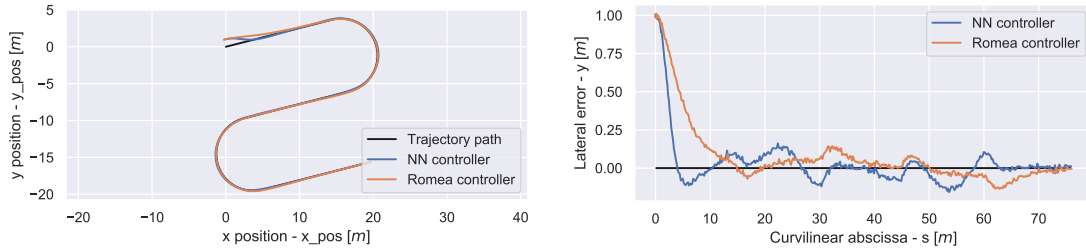


Figure 4.2: On the left: The *spline5* trajectory on a x, y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.

Figure 4.2 shows the reactivity of correcting the initial lateral error described previously, as the *NN controller* provides fast convergence to the trajectory, and tracking similar to the existing predictive controller, as shown in the error plot.

This result is clearly visible in the objective function and surface error plots over the curvilinear abscissa shown in Figure 4.3, as the *NN controller* avoids the large penalty at the start of the trajectory. However it seems that the *NN controller* is inducing large errors in the corners from $15 \leq s \leq 30$ and $45 \leq s \leq 60$.

Observing the steering over the curvilinear abscissa shown in Figure 4.4, the *NN controller's* control strategy seems to be based on a *bang-bang* control (as seen by the distinctive saw-tooth appearance of the steering state). This strategy may seem counter intuitive at first. But, due to the second order filter of the low-level steering controller, the control signal from the *NN controller* is being low-pass filtered. Unfortunately, this low-pass filter is not ideal as it still allows for some strong oscillations on the steering state. Furthermore, this makes the control task for the *NN controller* very complicated, as it is not predicting the steering value directly but

⁴This quick convergence to the trajectory seems to be a distinctive feature of all the NN based approaches.

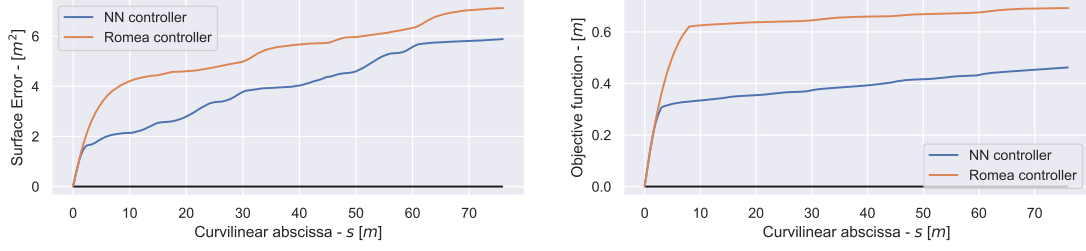


Figure 4.3: On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.

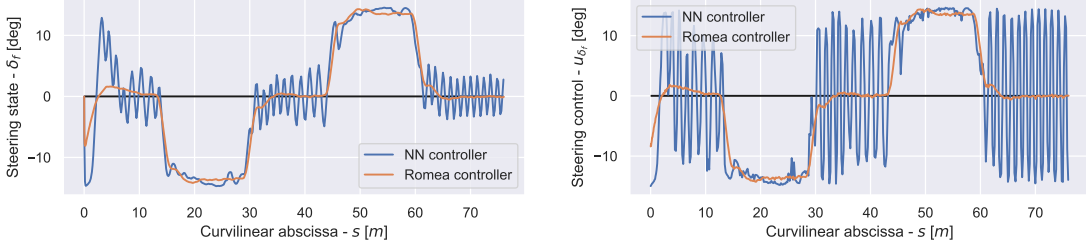


Figure 4.4: On the left: The real steering state over the curvilinear abscissa. On the right: The steering input from the controller over the curvilinear abscissa.

instead has found a pattern that allows for approximating the steering value through the non-ideal steering actuators. This could be solved though the tuning of the k_{steer} parameter of the objective function, however as it is based on the steering state and not the steering control it would only slightly dampen the oscillation visible on the steering control plot, and not remove them (the k_{steer} value is set correctly when use in other configurations, which implies this is a characteristic of this method of control). This makes the *NN controller* very opaque to understand. As such, a feature importance analysis may be performed in order to better understand the behavior of the neural network.

Feature importance

In order to better interpret and understand the neural network, a gradient based Feature importance analysis can be used to determine which inputs where useful, and quantify the utility of each input with respect to each output. See section A.4 for details on the theory and implementation of the gradient based Feature importance analysis for the neural network. Using the Feature importance analysis, the following results are obtained:

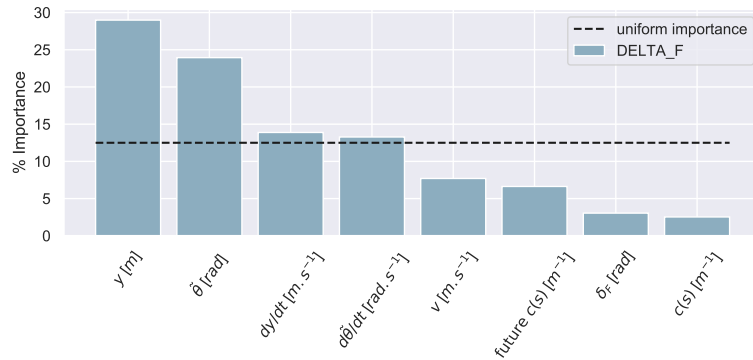


Figure 4.5: The feature importance for the *NN controller* method for each input, denoted in % of importance.

From the figure 4.5, the inputs that contribute the most the steering output of the method are

in order of importance the lateral error denoted y , the angular error denoted $\tilde{\theta}$, the rate of change of the lateral error denoted dy/dt , and the rate of change of the angular error denoted $d\tilde{\theta}/dt$ which all contribute a total of at least 80% of the variations of the steering output. The steering denoted δ_F and the immediate curvature denoted $c(s)$ are significantly below the expected importance for a uniform importance distribution, which implies that they are not significant for the output (5% total importance for both).

This shows that the *NN controller* is capable of imitating the existing methods for controlling the robotic platform without prior knowledge, as the Romea/ESBF methods also uses the lateral and angular error in order to accurately control the robot. Furthermore, the immediate curvature is not important to the *NN controller* when compared to the future curvature which has predictive capabilities over the errors, although it is a requirement for the objective function the trained neural network is able to achieve a tracking behavior using a quasi-PID approach.

Analysis of the approach

Overall, these results seem promising but are not ideal. Indeed, *NN controller* is able to learn a valid control law, but it is based on a highly oscillating control strategy, which will lead to dangerous behavior in real world conditions. Furthermore, it seems only able to outmatch the existing controllers due to their constant reactivity and that the *NN controller* can specialize its control output for the task and the robot. This non-ideal behavior may be due to:

- The objective function, but it is unclear how the objective function should be changed in this case.
- The dataset, but this is unlikely, as we are testing in the same training environment for a large amount of time, so overfitting is expected, not underfitting.
- The CMA-ES optimizer, but there seems to be no papers on the subject to indicate why or which optimizer would work better. Off empirical evidence, it seems that CMA-ES works best in our configuration (as shown in section A.3).

As such, there is no clear reason as to why the *NN controller* showed low performance, with the exception of the training difficulty. The training difficulty is a qualitative concept that defines how hard is it to train a valid function estimator (i.e a neural network) from the given target and setup. In this case, the neural network needs to estimate a valid control output, in order to minimize an objective function, using the errors and the curvatures. A task which requires a correct approximation of the robot's expected behavior for a given control input and state, which might be too complicated to learn and is only approximating it in order to approach a low enough objective value. But as seen, this can lead to unexpected output, such as the *bang-bang* control strategy.

This means, an alternative approach to controlling the robot's steering should be explored, in order to lower the training difficulty, and obtain better performance. An approach to this, would be combining both the existing control laws with a neural network, in order to augment the controllers. This serves two purposes: The first is improving the control law directly, meaning we should *at least* match the performance of the existing controllers. The second is to minimize the training difficulty, as this means the neural network would only need to correct or alter the existing control law, and not learn a control law from scratch.

Combining both the existing control laws with a neural network can be done in multiple ways. The first way that is explored in the next chapter is simply adding the output of the neural network, with the output of the existing control law, which can be seen as a corrective steering approach.

4.2 Corrective steer control [Delta NN ctrl]

As shown in the conclusion of the previous section, the replacement of existing control laws with a neural network can lead to unexpected side effect, and does not significantly improve the performance of the tracking task. As such, this chapter focuses on the hybridization of the control system, to include both the control law and a neural network in order to improve the overall performance of both. As with the previous chapter, the speed will be defined as a constant target speed, in order to compare the steering control independently to the speed control. The first

way for hybridization explored will be using the neural network's output as a corrective term for the steering angle, as detailed in the following section. Which will then be compared with the previously shown methods.

Experimental setup

Control loop setup

The neural network predicts a corrective steering term, which is added to the output of the existing control law, as shown in the figure 4.6 where the neural network takes the errors, curvature, and speed, then returns the predicted corrective steering control output which is added to the steering returned by the control law. The control law that is used in tandem is the Romea control law, as it had the lowest error of both model predictive controllers.⁵

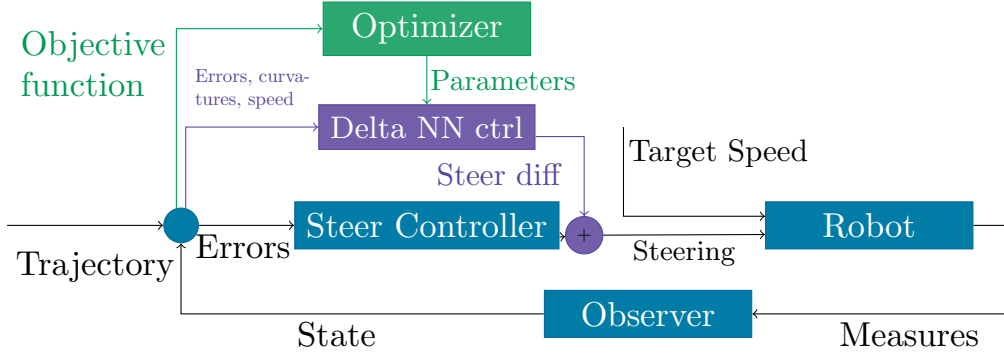


Figure 4.6: Overview of the proposed method.

The neural network is trained in a simulation using the CMA-ES optimizer depicted as the *Optimizer* in the figure 4.6, which takes the objective function value and returns the parameters. The neural network takes as input the same information as an existing steering controller, which is the lateral error, angular error, curvature, future curvature (20 sampled points over 5s horizon), speed, and the robot's steering state. And as control output the steering angle, with the speed control output being defined as constant. An Extended Kalman Filter (EKF) [53] is used as the *Observer* in order to filter the noise from the robot's sensors, and improve the accuracy of the tracking.

Metrics

The objective function used is identical to the previously defined first objective function, shown in the equation (4.3).

$$obj_1 = obj_{err} + k_{steer} obj_{steer}$$

Integrating both the neural network and a steering controller in parallel does not alter the training target, since both control tasks are the same.

The metric used in the analysis is identical to the previously defined surface error shown in the equation (4.4).

$$A_{error} = \sum_{i=0}^N \left| v_i \cos(\tilde{\theta}_i) \left(y_i + \frac{v_i \sin(\tilde{\theta}_i) \Delta t}{2} \right) \right| \Delta t$$

As it will allow for minimal bias when comparing the methods.

Training details

The neural network is trained over 5 unique trajectories (*estoril5*, *estoril7*, *estoril910*, *line*, and *spline5* as shown in A.12) twice with two varying scaling factors of 1 and 2 (this is done so longer trajectories are also tested with lower curvatures), at speeds of 1.0, 2.0, 3.0, and 4.0 $m.s^{-1}$, with varying grip conditions (cornering stiffness ranging from 30000 to 7000).

⁵It should be noted, this choice is arbitrary and that this method is not limited to the Romea controller.

Properties of the robuFAST experimental mobile platform depicted in section 5.3 are used as parameters: a wheelbase of $1.2m$, $430kg$ of weight, a max steering angle of 15° , and a max acceleration $0.5m.s^{-2}$

Simulated results

The trained method along with the baselines controllers, were tested in the simulation with the same speeds, trajectories, and grip conditions as used in the training of the trained method. These tests were run 100 times each in order to get a consistent mean and variation for each method.

The gains for the existing controllers are set to the same values defined in the section 4.1.

Quantitative Analysis

A first set of simulated runs was computed using the trajectories described in the appendix A.12. From this, the table 4.5 was obtained. It describes the Surface error from the equation (4.4), for

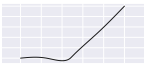



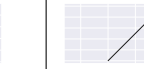
						
		estoril5	estoril7	estoril910	spline5	line
$1m.s^{-1}$	Romea	<u>1.18</u> (± 0.35)	1.40 (± 0.63)	29.19 (± 0.75)	2.37 (± 1.27)	<u>1.09</u> (± 0.04)
	EBSF	1.24 (± 0.37)	1.46 (± 0.70)	36.80 (± 0.65)	2.64 (± 1.16)	1.10 (± 0.04)
	<i>NN controller</i>	<u>1.43</u> (± 0.31)	2.42 (± 0.60)	21.63 (± 0.82)	3.94 (± 1.08)	1.29 (± 0.05)
	<i>Delta NN ctrl</i>	1.30 (± 0.22)	<u>1.32</u> (± 0.45)	<u>20.68</u> (± 1.02)	<u>2.01</u> (± 1.09)	1.48 (± 0.02)
$2m.s^{-1}$	Romea	<u>1.54</u> (± 0.50)	2.65 (± 1.26)	33.42 (± 0.79)	4.75 (± 1.56)	<u>1.12</u> (± 0.06)
	EBSF	<u>1.55</u> (± 0.50)	<u>2.31</u> (± 1.34)	39.75 (± 0.84)	5.76 (± 1.92)	<u>1.11</u> (± 0.06)
	<i>NN controller</i>	2.10 (± 0.33)	3.02 (± 1.14)	18.30 (± 0.74)	5.33 (± 1.22)	1.64 (± 0.06)
	<i>Delta NN ctrl</i>	2.00 (± 0.29)	2.48 (± 1.06)	<u>17.36</u> (± 0.65)	<u>4.36</u> (± 1.41)	1.68 (± 0.03)
$3m.s^{-1}$	Romea	3.13 (± 1.07)	5.96 (± 1.50)	53.41 (± 1.40)	9.59 (± 1.27)	<u>1.21</u> (± 0.10)
	EBSF	<u>2.09</u> (± 0.70)	3.54 (± 1.35)	47.21 (± 1.18)	14.88 (± 1.91)	<u>1.16</u> (± 0.07)
	<i>NN controller</i>	3.19 (± 0.52)	3.55 (± 1.20)	20.59 (± 1.25)	<u>5.78</u> (± 1.31)	2.49 (± 0.11)
	<i>Delta NN ctrl</i>	2.47 (± 0.51)	<u>3.31</u> (± 1.09)	<u>17.97</u> (± 0.71)	6.16 (± 1.56)	1.27 (± 0.07)
$4m.s^{-1}$	Romea	4.31 (± 1.49)	7.90 (± 1.49)	55.82 (± 2.54)	11.29 (± 1.48)	<u>1.28</u> (± 0.17)
	EBSF	<u>3.59</u> (± 0.92)	<u>4.96</u> (± 1.02)	51.97 (± 2.33)	92.30 (± 176.29)	2.15 (± 0.25)
	<i>NN controller</i>	6.60 (± 1.42)	7.90 (± 1.97)	28.53 (± 3.25)	<u>8.80</u> (± 2.71)	4.87 (± 0.85)
	<i>Delta NN ctrl</i>	4.85 (± 0.90)	6.88 (± 1.13)	<u>25.12</u> (± 1.33)	9.55 (± 1.91)	2.12 (± 0.08)

Table 4.5: Surface error in $[m^2]$ of each method at all the speeds and trajectories used during training, with an **initial error of $0m$** .

each method at all the speeds and trajectories used during the training, with an initial error of $0m$. The underlined and bold values mean that the result is significant and has a p-value below 10^{-3} , determined using the *Welch-t test* [81].

Overall, the average surface error for the *Delta NN ctrl* was noticeably lower than the previous methods at $5.79m^2$, where as the surface error for Romea was $8.39m^2$ (a 31.0% reduction), and the surface error for *NN controller* was $6.69m^2$ (a 13.4% reduction). From this table, more specific strengths and weaknesses can be observed, the *Delta NN ctrl* was able to match or exceed the performance of the existing controllers in some cases. Notably the *estoril7*, *estoril910*, and the *spline5* trajectories, where as it seems to struggle with *estoril5* and the *line* trajectories. Furthermore, it seems that the *NN controller* will outperform the *Delta NN ctrl* method on the *spline5* trajectory at higher speed, but overall it seems that the *Delta NN ctrl* method is an iterative improvement over the *NN controller* method as it is able to match or exceed the performance of the Romea controller over the *estoril7* and *estoril5* trajectories. As with the *NN controller* method, when adding an initial lateral error of $1m$ at the start of the trajectories, the *Delta NN ctrl* method is capable of impressive performance, as seen in table 4.6.

Overall, the average surface error for the *Delta NN ctrl* was noticeably lower than the previous methods at $7.47m^2$, where as the surface error for Romea was $14.2m^2$ (a 47.6% reduction), and the surface error for *NN controller* was $8.83m^2$ (a 15.4% reduction). From this table, more specific

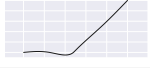



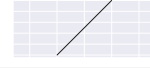
						
		estoril5	estoril7	estoril910	spline5	line
$1m.s^{-1}$	Romea	4.42 (± 0.36)	4.66 (± 0.61)	32.44 (± 0.76)	5.61 (± 1.25)	4.35 (± 0.08)
	EBSF	6.19 (± 0.37)	6.39 (± 0.69)	41.76 (± 0.65)	7.58 (± 1.13)	6.04 (± 0.06)
	<i>NN controller</i>	3.59 (± 0.31)	4.55 (± 0.59)	23.75 (± 0.81)	6.07 (± 1.06)	3.42 (± 0.10)
	<i>Delta NN ctrl</i>	3.07 (± 0.21)	3.09 (± 0.45)	22.44 (± 1.03)	3.78 (± 1.07)	3.26 (± 0.10)
$2m.s^{-1}$	Romea	5.92 (± 0.49)	7.07 (± 1.25)	37.81 (± 0.79)	9.03 (± 1.55)	5.52 (± 0.09)
	EBSF	6.32 (± 0.49)	7.07 (± 1.32)	44.48 (± 0.83)	10.46 (± 1.92)	5.88 (± 0.09)
	<i>NN controller</i>	4.30 (± 0.32)	5.12 (± 1.14)	20.44 (± 0.72)	7.44 (± 1.20)	3.67 (± 0.06)
	<i>Delta NN ctrl</i>	3.65 (± 0.29)	4.14 (± 1.06)	19.02 (± 0.66)	5.95 (± 1.40)	3.31 (± 0.08)
$3m.s^{-1}$	Romea	11.19 (± 1.11)	13.81 (± 1.53)	61.53 (± 1.40)	16.54 (± 1.27)	9.21 (± 0.14)
	EBSF	7.75 (± 0.72)	9.15 (± 1.33)	52.84 (± 1.16)	20.26 (± 1.87)	6.80 (± 0.11)
	<i>NN controller</i>	5.44 (± 0.50)	5.81 (± 1.25)	22.64 (± 1.19)	7.80 (± 1.27)	4.59 (± 0.13)
	<i>Delta NN ctrl</i>	4.12 (± 0.48)	4.97 (± 1.09)	19.61 (± 0.69)	7.79 (± 1.57)	2.92 (± 0.20)
$4m.s^{-1}$	Romea	12.28 (± 1.56)	15.62 (± 1.52)	63.83 (± 2.54)	18.37 (± 1.43)	9.15 (± 0.17)
	EBSF	9.18 (± 0.96)	10.48 (± 0.98)	57.62 (± 2.46)	91.77 (± 165.52)	7.69 (± 0.23)
	<i>NN controller</i>	8.88 (± 1.41)	9.94 (± 1.78)	31.63 (± 5.14)	10.47 (± 2.59)	7.19 (± 0.92)
	<i>Delta NN ctrl</i>	6.50 (± 0.94)	8.51 (± 1.17)	26.77 (± 1.33)	11.15 (± 2.00)	3.73 (± 0.09)

Table 4.6: Surface error in $[m^2]$ of each method at all the speeds and trajectories used during training, with an **initial error of 1m**.

strengths and weaknesses can be observed, the *Delta NN ctrl* was able to exceed the performance of the existing controllers in all of the tested cases, with the exception of the *line* trajectory which can be explained as the neural network over-reacting to the system noise, when a constant steering of 0° is required to get the best score. Furthermore, the *Delta NN ctrl* consistently outmatched the *NN controller*, with the exception once more of the *spline5* trajectory at high speeds.

Qualitative Analysis

When focusing with a qualitative analysis over the *spline5* trajectory, at $2m.s^{-1}$ with 1m of initial error, the following results are obtained. Figure 4.7 shows the same behavior as previously

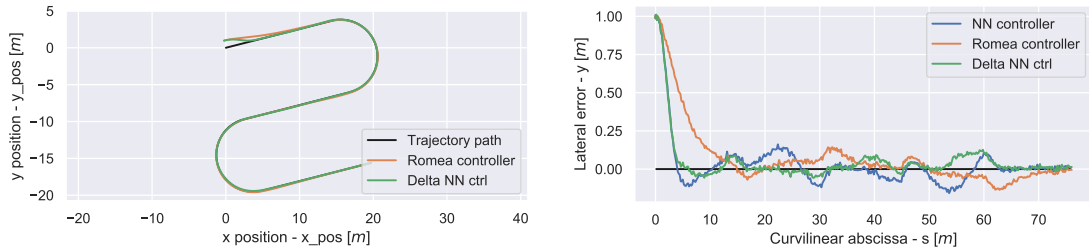


Figure 4.7: On the left: The *spline5* trajectory on a x, y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.

described. The reactivity of the initial lateral error is very high, as the *Delta NN ctrl* quickly converges to the trajectory, and is able to reduce the lateral error once converged when compared to the Romea controller.

This result is quite clear on the objective function and surface error from figure 4.8, as the *Delta NN ctrl* is able to minimize the large initial penalty, all while preserving a low rate of change of said objective function and surface error, which implies a low overall error after convergence.

The *NN controller* showed some worrying behavior on the steering angle, which is reduced when the neural network is used as a corrective method, rather than a replacement to existing control laws. This implies that the reduction of training difficulty inherit to this method, was a determining factor in the performance of the trained neural network. Indeed, as shown in figure 4.9

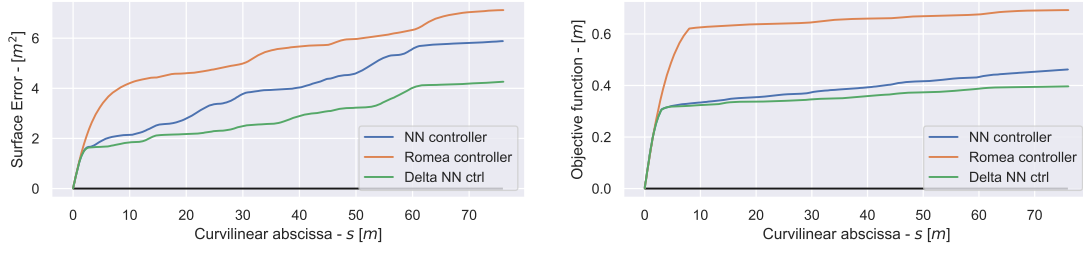


Figure 4.8: On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.

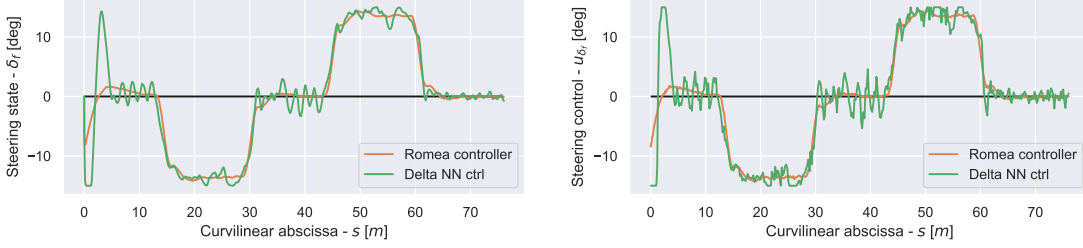


Figure 4.9: On the left: The real steering state over the curvilinear abscissa. On the right: The steering input from the controller over the curvilinear abscissa.

the capacity of the neural network to correct the initial error from $0 \leq s \leq 5$, while not resorting to a *bang-bang* control strategy as seen by the *NN controller*, along with the lower overall error shows that the resulting neural network is able to reach high performance compared to the *NN controller*.

Feature importance

In order to better interpret and understand the neural network, a gradient based Feature importance analysis can be used to determine which inputs were useful, and quantify the utility of each input with respect to each output. See section A.4 for details on the theory and implementation of the gradient based Feature importance analysis for the neural network. Using the Feature importance analysis, the following results are obtained:

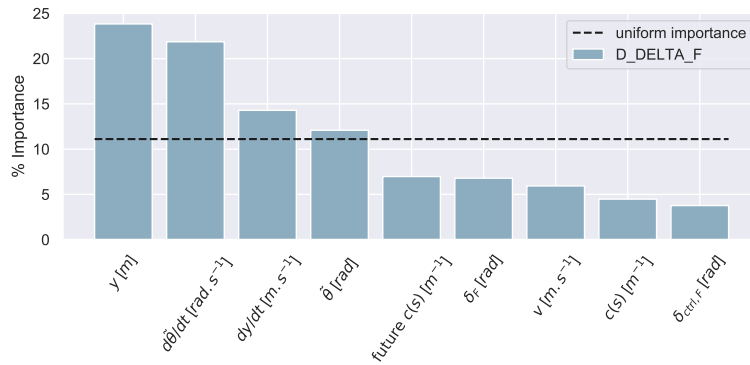


Figure 4.10: The feature importance for the *Delta NN ctrl* method for each input, denoted in % of importance.

From the figure 4.10, the inputs that contribute the most the corrective steering output of the method are in order of importance the lateral error denoted y , the rate of change of the angular error denoted $d\tilde{\theta}/dt$, the rate of change of the lateral error denoted dy/dt , the angular error denoted $\tilde{\theta}$, and the future curvature denoted future $c(s)$ which all contribute a total of at least 80% of the variations of the corrective steering output. The control steering output denoted

$\delta_{ctrl,F}$ is significantly below the expected importance for a uniform importance distribution, which surprisingly implies that it is not significant for the output. This can be explained as the neural network using the other inputs in order to predict the expected output of the control law.

This shows that the *Delta NN ctrl* is correcting the control output in a similar manner to the *NN controller*, with a higher emphasis on the rate of change of the errors and the future curvature. This implies that the *Delta NN ctrl* is using these inputs in order to better adapt the control output of the existing controller, as the neural network has determined that they are underused by the controller law due to its inclusion even though the existing controller is already exploiting them.

Analysis of the approach

Overall the *Delta NN ctrl* has shown promising results, as it is able to obtain a better behavior and higher performance using this configuration, when compared to the previous *NN controller*. Indeed, it is able to follow the same trajectories as the MPC alone, with similar performance while being able to adapt the control output in order to compensate for strong initial errors, and to lower the overall error.

As such, this configuration of corrective steering is indeed a promising method. However, it is not the only method of adapting a control law using a neural network.

4.3 Online control parameter tuning for existing steer controller [*NN gain tuner*]

The controllers are configured using their control parameters, which often depend on the changes in the set-points and environment such as the trajectory, including speed, for path following tasks.



Figure 4.11: Example of sources of influence on the optimal control parameters. Left: wheel, actuator dynamics. Middle: GPS sensor, perception quality. Right: ground, environment.

It will also depend on the perception quality (such as a GPS or a LIDAR accuracy, figure 4.11), and it will depend on the highly dynamic nature of the system (such as the actuator properties) as much as on the environment characteristics (such as the tyre-ground interface, figure 4.11). All these aspects must be taken into account for an ideal performance, when the environment is changing over time. As an initial example, the NN gain ctrl method will only be given the same inputs as the previous example in order to preserve the comparability with the previous methods.

When comparing with *Delta NN ctrl*, the NN gain ctrl method has a few key differences.

- Preserving existing systems: The system can remain the same, but the controller's behavior is altered in real time in order to improve its performance and adaptability. As such, it can be added as a "drop-in" improvement to existing in-situ control system.
- The neural network no longer needs to predict the controller: This lowers even further the training difficulty, as the expected behavior of the controller on the system is no longer needed in order to determine an accurate corrective term. Whereas the parameter tuning approach requires only the expected behavior of the system and the controller together, which is often simpler.

- Isolation of the neural network from the control output: this approach will not require direct control of the robot, which means that the failure cases are reduced substantially and the predictability of the system increases as the potential impact of the neural network to the system is theoretically reduced. However, by removing direct control to the steering control output, this method might sacrificing potential performance, as the controller might not be able to reach a desired configuration from tuning the control parameters alone.
- Permitting deterministic behavior: By only modulating the control parameters through a neural network, a high degree of explainability and deterministic behavior can be preserved, which can be an important consideration for industrial use or certification of the system.
- Visible failure case: The range of valid control parameters can be estimated, meaning a failure case where the neural network predicts invalid control parameters can be foreseen and corrected, preventing potential loss of control in safety sensitive systems.

Before tuning the control parameters, an understanding of what control parameters are needed in order to correctly interpret and analyze the behavior of the robot from modulating the control parameters. A brief explanation of control parameters and control parameters tuning is as such described in the following section.

Control parameter tuning

Control parameters

Integrating the full state of the system and all the sensor information into a control law is not a simple task, as it can be relatively hard to qualify how a controller should react to the perceived information in full detail. Usually, controllers are tuned for a given environmental state, and this tuning encodes the unmodeled aspects of the controller. This tuning is defined as the control parameters of a controller.

The control parameters are values that define the reactivity of the controller to specific state variables. They are defined as the control effort relative to the error, in terms of time or distance to convergence. They are usually set so the controller is critically damped, and so that the controller will quickly converge to the set point, but not overshoot too much or oscillate.

With this, ideal control parameters would then tune the controller to obtain a fast convergence to the set point, a non-oscillatory control, and to minimize the control errors overall.

Constant control parameters

Finding the ideal control parameters has been a major subject of research in optimal control theory. As such many methods exist in order to find the optimal fixed control parameters for a given environment, here are a few examples:

- Empirical tuning by hand, where an expert tunes the controller by trial and error, in order to obtain the desired behavior.
- Algorithmic methods (e.g.: Ziegler–Nichols), where an algorithm breaks down the tuning into simpler step-by-step adjustments, until the desired behavior is obtained [82].
- A black box optimizer in a simulation, where the ideal tuning is obtained when a target metric has reached its local minimum [83].

However as described previously, the optimal control parameters will depend on the changes in the environment, as such a dynamic parameter tuning method will be used.

Dynamic control parameters

The adjustment of the control parameters, in real time over the course of the control task, will be called dynamic control parameters tuning. For this, two common classes of methods exist.

The first is fuzzy logic control parameters scheduling, the second is LQR (Linear Quadratic Regulator) control parameters optimization.

Fuzzy logic control parameters scheduling [84, 85, 86], is a method that determines when a change in control parameters should occur, then once a change occurs it changes the current control parameters to a predefined parameters for that environmental state. This method implies that all the environmental states have been mapped to the fuzzy logic, and that each control parameter has been tuned for that specific environmental state. This task can increase time used for the control parameters tuning task by an order of magnitude; as multiple fixed control parameters need to be tuned for each possible environmental state.

LQR control parameters optimization [87, 88] uses a linear-quadratic regulator with a model of the environment, in order to tune the control parameters in real time. This method seems like an ideal contender as it can determine in real time the ideal control parameters. However, the drawback of this system is that the model of the environment that is used must be very well defined in order to get an accurate enough prediction. Furthermore, the LQR method depends on linearization of the model over future timesteps, which can be both inaccurate and computationally expensive.

A zoo of control parameters tuning methods have been proposed. However the key aspect that is targeted is the adaptability of the control parameters tuning, while keeping the controller explainable. This is unfortunately a trade-off between explainability vs adaptability:

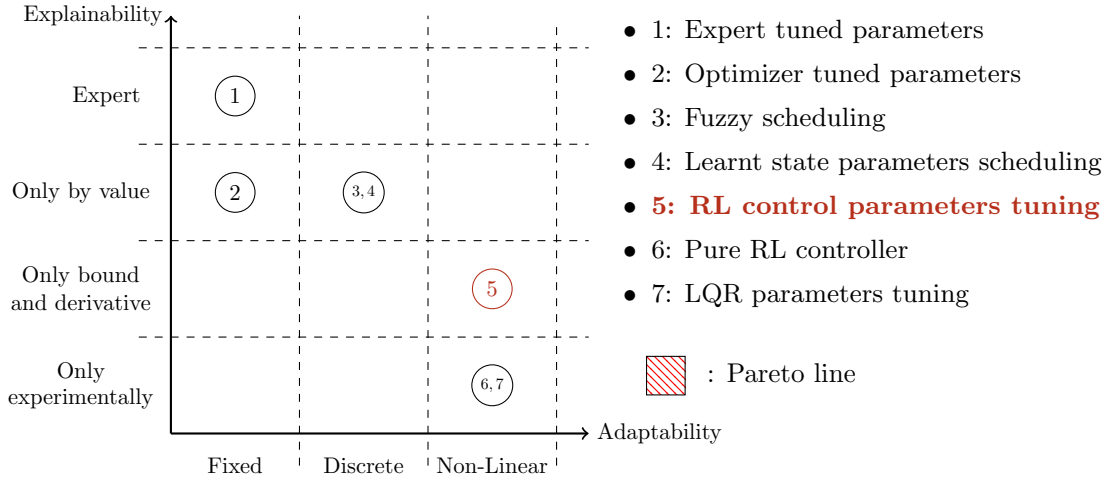


Figure 4.12: Explainability and adaptability compromise in control parameter tuning for controllers.

The trade-off can be seen on the figure 4.12, with the following definition for each class of methods:

- Expert tuned parameters: A constant control parameters, tuned by an expert for a given task. Explainable by the expert.
- Optimizer tuned parameters: A constant control parameters, tuned by an optimization algorithm for a given task [83]. Explainable by analyzing the behavior of the control parameters in the system.
- Fuzzy scheduling: A set of constant control parameters, changing according to the environment using fuzzy logic [84, 85, 86]. Explainable by analyzing the behavior of each control parameters in the system.
- Learned state parameters scheduling: A set of constant control parameters, changing according to the environment using a machine learning method such as DQN [19]. Explainable by analyzing the behavior of each control parameters in the system.
- RL control parameters tuning: Reinforcement learning algorithm to tune control parameters depending on the state (as depicted in section 4.3). Explainable through the bounds and the variations of the control parameters.
- Pure RL controller: Replace the control system by a reinforcement learning algorithm. Explainable through experimentation alone (as depicted in section 4.1 and in [24]).

- LQR parameters tuning: linear-quadratic regulator to tune control parameters depending on the state and the predicted future state [87, 88]. Explainable through experimentation alone.

Here, a continuous control parameters output is considered more adaptable than discrete values, as the continuous methods used to output the control parameters will try and generalize in between the two discrete values that a discrete method would output. This implies a more adaptable and continuous transition between two control parameters.

These aspects show that using a neural network trained using reinforcement learning is valid for a continuous real-time controller parameter tuning method. As such, the following sections describe the application of the neural network as a controller parameter tuning method, and the performance of said method when compared to the previous methods shown.

Experimental setup

Control loop setup

The neural network predicts the control parameters in real-time. In this case they are the steering gains and steering horizon, which are then passed to the controller before the controller calculates the steering angle. As shown in the figure 4.13 where the neural network takes the errors, curvature, and speed, then returns the steering gains and steering horizon. The control law that is used in tandem is the Romea control law, in order to preserve the comparability with the previous methods.

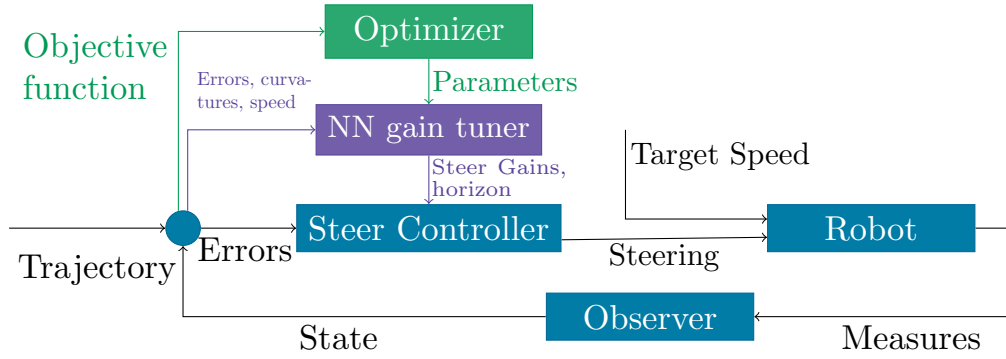


Figure 4.13: Overview of the proposed method.

The neural network is trained in a simulation using the CMA-ES optimizer depicted as the *Optimizer* in the figure 4.13, which takes the objective function value and returns the parameters. The neural network takes as input the same information as an existing steering controller, which is the lateral error, angular error, curvature, future curvature (20 sampled points over 5s horizon), speed, and the robot's steering state. And it outputs the control parameters. The speed is defined as constant before the path following. An Extended Kalman Filter (EKF) [53] is used as the *Observer* in order to filter the noise from the robot's sensors, and improve the accuracy of the tracking.

Metrics

The objective function used is identical to the previously defined first objective function, shown in the equation (4.3).

$$obj_1 = obj_{err} + k_{steer} obj_{steer}$$

Integrating the neural network as a control parameter tuner for the steering controller does not alter the training target, since both control tasks are the same, which is to steer the robot.

The metric used in the analysis is identical to the previously defined surface error shown in the equation (4.4).

$$A_{error} = \sum_{i=0}^N \left| v_i \cos(\tilde{\theta}_i) \left(y_i + \frac{v_i \sin(\tilde{\theta}_i) \Delta t}{2} \right) \right| \Delta t$$

As it will allow for minimal bias when comparing the methods.

Training details

The neural network is trained over 5 unique trajectories (*estoril5*, *estoril7*, *estoril910*, *line*, and *spline5*) as shown in A.12) twice with two varying scaling factors of 1 and 2 (this is done so longer trajectories are also tested with lower curvatures), at speeds of 1.0, 2.0, 3.0, and 4.0 $m.s^{-1}$, with varying grip conditions (cornering stiffness ranging from 30000 to 7000).

Properties of the robuFAST experimental mobile platform are used as parameters: a wheelbase of 1.2m, 430kg of weight, a max steering angle of 15° , and a max acceleration $0.5m.s^{-2}$

Simulated results

The trained method along with the baselines controllers, were tested in the simulation with the same speeds, trajectories, and grip conditions as used in the training of the trained method. These tests were run 100 times each in order to get a consistent mean and variation for each method.

The gains for the existing controllers are set to the same values defined in the section 4.1.

Quantitative Analysis

A first set of simulated runs was computed using the trajectories described in the appendix A.12. From this, the table 4.7 was obtained. It describes the surface error from the equation (4.4), for

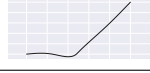
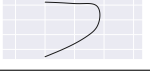
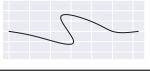


						
		estoril5	estoril7	estoril910	spline5	line
$1m.s^{-1}$	Romea	1.18 (± 0.35)	1.40 (± 0.63)	29.19 (± 0.75)	2.37 (± 1.27)	<u>1.09</u> (± 0.04)
	EBSF	1.24 (± 0.37)	1.46 (± 0.70)	36.80 (± 0.65)	2.64 (± 1.16)	1.10 (± 0.04)
	<i>NN controller</i>	1.43 (± 0.31)	2.42 (± 0.60)	21.63 (± 0.82)	3.94 (± 1.08)	1.29 (± 0.05)
	<i>Delta NN ctrl</i>	1.30 (± 0.22)	1.32 (± 0.45)	20.68 (± 1.02)	2.01 (± 1.09)	1.48 (± 0.02)
	<i>NN gain tuner</i>	1.16 (± 0.23)	1.37 (± 0.51)	19.78 (± 1.97)	2.00 (± 1.29)	1.17 (± 0.03)
$2m.s^{-1}$	Romea	1.54 (± 0.50)	2.65 (± 1.26)	33.42 (± 0.79)	4.75 (± 1.56)	<u>1.12</u> (± 0.06)
	EBSF	1.55 (± 0.50)	2.31 (± 1.34)	39.75 (± 0.84)	5.76 (± 1.92)	<u>1.11</u> (± 0.06)
	<i>NN controller</i>	2.10 (± 0.33)	3.02 (± 1.14)	18.30 (± 0.74)	5.33 (± 1.22)	1.64 (± 0.06)
	<i>Delta NN ctrl</i>	2.00 (± 0.29)	2.48 (± 1.06)	17.36 (± 0.65)	4.36 (± 1.41)	1.68 (± 0.03)
	<i>NN gain tuner</i>	1.54 (± 0.31)	2.08 (± 1.07)	16.99 (± 1.46)	3.21 (± 1.57)	1.32 (± 0.06)
$3m.s^{-1}$	Romea	3.13 (± 1.07)	5.96 (± 1.50)	53.41 (± 1.40)	9.59 (± 1.27)	<u>1.21</u> (± 0.10)
	EBSF	2.09 (± 0.70)	3.54 (± 1.35)	47.21 (± 1.18)	14.88 (± 1.91)	<u>1.16</u> (± 0.07)
	<i>NN controller</i>	3.19 (± 0.52)	3.55 (± 1.20)	20.59 (± 1.25)	5.78 (± 1.31)	2.49 (± 0.11)
	<i>Delta NN ctrl</i>	2.47 (± 0.51)	3.31 (± 1.09)	17.97 (± 0.71)	6.16 (± 1.56)	1.27 (± 0.07)
	<i>NN gain tuner</i>	2.19 (± 0.52)	3.00 (± 1.32)	19.33 (± 1.34)	5.49 (± 2.87)	1.45 (± 0.11)
$4m.s^{-1}$	Romea	4.31 (± 1.49)	7.90 (± 1.49)	55.82 (± 2.54)	11.29 (± 1.48)	<u>1.28</u> (± 0.17)
	EBSF	3.59 (± 0.92)	4.96 (± 1.02)	51.97 (± 2.33)	92.30 (± 176.29)	<u>2.15</u> (± 0.25)
	<i>NN controller</i>	6.60 (± 1.42)	7.90 (± 1.97)	28.53 (± 3.25)	8.80 (± 2.71)	4.87 (± 0.85)
	<i>Delta NN ctrl</i>	4.85 (± 0.90)	6.88 (± 1.13)	25.12 (± 1.33)	9.55 (± 1.91)	2.12 (± 0.08)
	<i>NN gain tuner</i>	3.58 (± 1.24)	4.89 (± 1.81)	26.43 (± 4.40)	10.17 (± 6.09)	1.76 (± 0.18)

Table 4.7: Surface error in $[m^2]$ of each method at all the speeds and trajectories used during training, with an **initial error of 0m**.

each method at all the speeds and trajectories used during the training, with an initial error of 0m. The underlined and bold values mean that the result is significant and has a p-value below 10^{-3} , determined using the *Welch-t test* [81].

Overall, the average surface error for the *NN gain tuner* was noticeably lower than the previous methods at $4.98m^2$, where as the surface error for Romea was $8.39m^2$ (a 40.7% reduction), the surface error for *NN controller* was $6.69m^2$ (a 25.6% reduction), and the surface error for *Delta NN ctrl* was $5.79m^2$ (a 14.0% reduction). From this table, more specific strengths and weaknesses can be observed, the *NN gain tuner* was able to match or exceed the performance of the existing controllers in most cases. Notably the *estoril7*, *estoril910*, *spline5*, and the *estoril5* trajectories, where as it seems to struggle with the *line* trajectory as did the previous methods.

The differences over the *line* trajectory can be interpreted due to the sensitivity of the proposed methods with respect to the noise, indeed the *line* trajectory is a simple task (requiring that only that $\delta_f = 0$ in order to be solved optimally), and as such any noise would be overcompensated which degrades the overall performance. Furthermore, it seems that the *NN gain tuner* will outperform the *Delta NN ctrl* and *NN controller* methods on the *estoril5* and *estoril7* trajectories, but remain comparable on the *estoril910* and *spline5* trajectories. Overall it seems that the *NN gain tuner* method is an iterative improvement over the *NN controller* method, and is comparable to the *Delta NN ctrl* as it is able to match or exceed the performance of the Romea controller. As with the *NN controller* and *Delta NN ctrl* methods, when adding an initial lateral error of 1m at the start of the trajectories, the *NN gain tuner* method is capable of impressive performance, as seen in table 4.8.

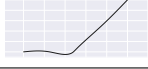

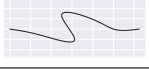

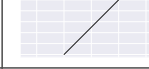
						
		estoril5	estoril7	estoril910	spline5	line
$1m.s^{-1}$	Romea	4.42 (± 0.36)	4.66 (± 0.61)	32.44 (± 0.76)	5.61 (± 1.25)	4.35 (± 0.08)
	EBSF	6.19 (± 0.37)	6.39 (± 0.69)	41.76 (± 0.65)	7.58 (± 1.13)	6.04 (± 0.06)
	<i>NN controller</i>	3.59 (± 0.31)	4.55 (± 0.59)	23.75 (± 0.81)	6.07 (± 1.06)	3.42 (± 0.10)
	<i>Delta NN ctrl</i>	3.07 (± 0.21)	3.09 (± 0.45)	22.44 (± 1.03)	3.78 (± 1.07)	3.26 (± 0.10)
	<i>NN gain tuner</i>	2.98 (± 0.23)	3.19 (± 0.51)	21.59 (± 1.98)	3.83 (± 1.26)	3.00 (± 0.10)
$2m.s^{-1}$	Romea	5.92 (± 0.49)	7.07 (± 1.25)	37.81 (± 0.79)	9.03 (± 1.55)	5.52 (± 0.09)
	EBSF	6.32 (± 0.49)	7.07 (± 1.32)	44.48 (± 0.83)	10.46 (± 1.92)	5.88 (± 0.09)
	<i>NN controller</i>	4.30 (± 0.32)	5.12 (± 1.14)	20.44 (± 0.72)	7.44 (± 1.20)	3.67 (± 0.06)
	<i>Delta NN ctrl</i>	3.65 (± 0.29)	4.14 (± 1.06)	19.02 (± 0.66)	5.95 (± 1.40)	3.31 (± 0.08)
	<i>NN gain tuner</i>	3.21 (± 0.31)	3.78 (± 1.06)	18.68 (± 1.51)	5.14 (± 1.57)	3.05 (± 0.10)
$3m.s^{-1}$	Romea	11.19 (± 1.11)	13.81 (± 1.53)	61.53 (± 1.40)	16.54 (± 1.27)	9.21 (± 0.14)
	EBSF	7.75 (± 0.72)	9.15 (± 1.33)	52.84 (± 1.16)	20.26 (± 1.87)	6.80 (± 0.11)
	<i>NN controller</i>	5.44 (± 0.50)	5.81 (± 1.25)	22.64 (± 1.19)	7.80 (± 1.27)	4.59 (± 0.13)
	<i>Delta NN ctrl</i>	4.12 (± 0.48)	4.97 (± 1.09)	19.61 (± 0.69)	7.79 (± 1.57)	2.92 (± 0.20)
	<i>NN gain tuner</i>	3.89 (± 0.54)	4.76 (± 1.32)	20.92 (± 1.40)	7.28 (± 2.97)	3.18 (± 0.23)
$4m.s^{-1}$	Romea	12.28 (± 1.56)	15.62 (± 1.52)	63.83 (± 2.54)	18.37 (± 1.43)	9.15 (± 0.17)
	EBSF	9.18 (± 0.96)	10.48 (± 0.98)	57.62 (± 2.46)	91.77 (± 165.52)	7.69 (± 0.23)
	<i>NN controller</i>	8.88 (± 1.41)	9.94 (± 1.78)	31.63 (± 5.14)	10.47 (± 2.59)	7.19 (± 0.92)
	<i>Delta NN ctrl</i>	6.50 (± 0.94)	8.51 (± 1.17)	26.77 (± 1.33)	11.15 (± 2.00)	3.73 (± 0.09)
	<i>NN gain tuner</i>	5.26 (± 1.30)	6.60 (± 1.82)	27.40 (± 3.52)	11.56 (± 5.80)	3.44 (± 0.20)

Table 4.8: Surface error in $[m^2]$ of each method at all the speeds and trajectories used during training, with an **initial error of 1m**.

Overall, the average surface error for the *NN gain tuner* was noticeably lower than the previous methods at $6.70m^2$, where as the surface error for Romea was $14.2m^2$ (a 53.0% reduction), the surface error for *NN controller* was $8.83m^2$ (a 24.1% reduction), and the surface error for *Delta NN ctrl* was $7.47m^2$ (a 10.3% reduction). From this table, more specific strengths and weaknesses can be observed, the *NN gain tuner* was able to exceed the performance of the existing controllers in all of the tested cases. Furthermore, the *NN gain tuner* consistently outmatched the previous methods over the *estoril5* and *estoril7* trajectories, while being comparable to Detla NN ctrl over the *estoril910*, *spline5*, and *line* trajectories.

Qualitative Analysis

When focusing with a qualitative analysis over the *spline5* trajectory, at $2m.s^{-1}$ with 1m of initial error, the following results are obtained. Figure 4.14 shows the same behavior as previously described. The reactivity of the initial lateral error is very high, as the *NN gain tuner* quickly converges to the trajectory, and is able to significantly reduce the lateral error once converged when compared to the previously described methods.

This result is quite clear on the objective function and surface error from figure 4.15, as the *NN gain tuner* is able to minimize the large initial penalty, all while obtaining a very low rate

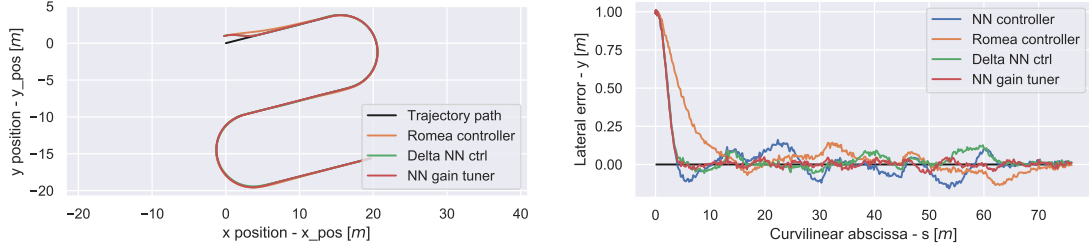


Figure 4.14: On the left: The *spline5* trajectory on a x, y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.

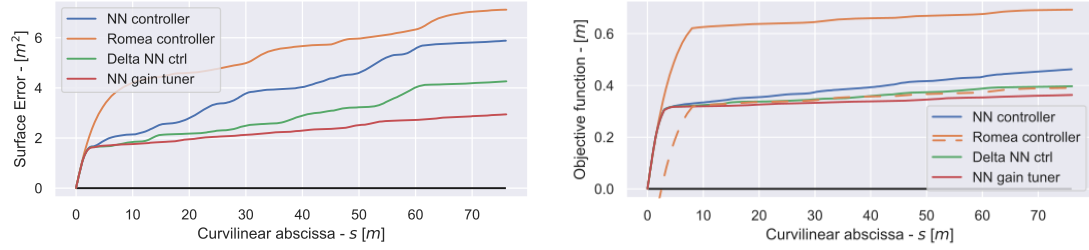


Figure 4.15: On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.

of change of said objective function and surface error, which implies a very low overall error after convergence. The Romea controller is not very comparable on this figure due to the difference in the initial error, however it's rate of change seems close to the *NN gain tuner* method (as shown in the orange dashed line on figure 4.15), and indeed this is confirmed with the surface error, where the Romea controller has a slightly higher rate of change of the error when compared to the *NN gain tuner*, which implies that without the initial error, the *NN gain tuner* would likely outperform the other methods.

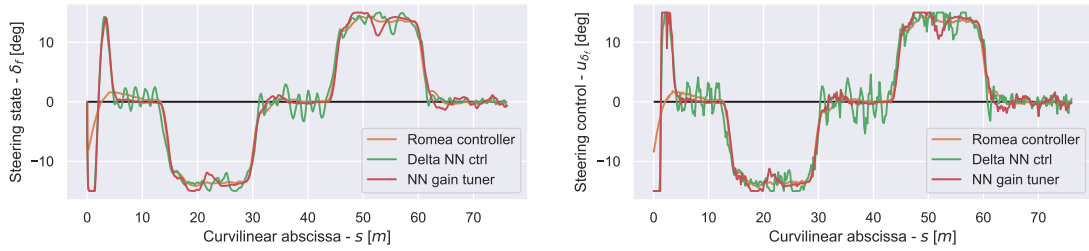


Figure 4.16: On the left: The real steering state over the curvilinear abscissa. On the right: The steering input from the controller over the curvilinear abscissa.

The previously shown methods had some oscillatory behavior on the steering angle. However, as illustrated in figure 4.16 the steering controller's output is very stable when a neural network is used to tune the control parameters in real time, when compared to the *Delta NN ctrl*. Furthermore, the modulation of the control parameters allows for the controller to quickly converge to the desired setpoint, similarly to when the neural network was modulating the steering output directly. This shows that for this controller, the control parameter tuning can allow for similar corrective behavior as seen previously, all while being isolated from the direct steering control.

As the neural network is only modulating the control parameters, this means that the desired behavior is encoded in said control parameters. Figure 4.17 shows these control parameters.

First for the control gains, the gains seem very high initially due to the initial lateral error, with increases visible at each sharp transition of the curvature at $15 \leq s \leq 25$, $30 \leq s \leq 35$, $45 \leq s \leq 55$, and $60 \leq s \leq 65$ in order to keep the lateral error low during said corners. Furthermore, the

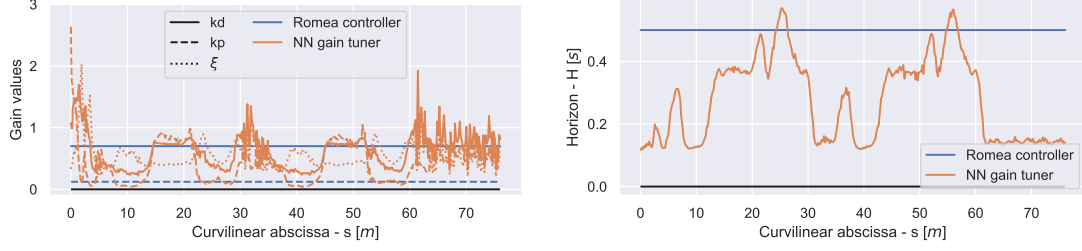


Figure 4.17: On the left: The gains over the curvilinear abscissa. On the right: The horizon over the curvilinear abscissa.

dampening of the controller denoted ξ is also visible, where the dampening is quite low in general at around 0.5, except during transitional states such as the corners and initial error, which seems to suggest that the neural network is increasing the reactivity of the control law during the stable states, while reducing it when it could lead to oscillatory or dangerous behavior. This allows the neural network and controller to correct any minor errors during the steady states very quickly, while still being stable and smooth when a transition occurs.

The second aspect of the control parameters is the horizon of the predictive section of the controller. Figure 4.17 shows that the horizon is quite low in general, implying that the neural network does not want the controller to react too early to the curvature. However, when a corner approaches, the neural network substantially increases the horizon at $s = 12$ and $s = 42$. Interestingly this happens to be approximately $3m$ before the first and second corner respectively which shows that the neural network is not overriding the predictive nature of the controller. It can be supposed that the neural network is reducing the horizon when it is not necessary, in order to prevent the controller reacting too strongly to any noise or abnormalities in the curvature, which allows for higher control gains when possible, and as such an overall better path following strategy.

Feature importance

In order to better interpret and understand the neural network, a gradient based Feature importance analysis can be used to determine which inputs were useful, and quantify the utility of each input with respect to each output. See section A.4 for details on the theory and implementation of the gradient based Feature importance analysis for the neural network. Using the Feature importance analysis, the following results are obtained:

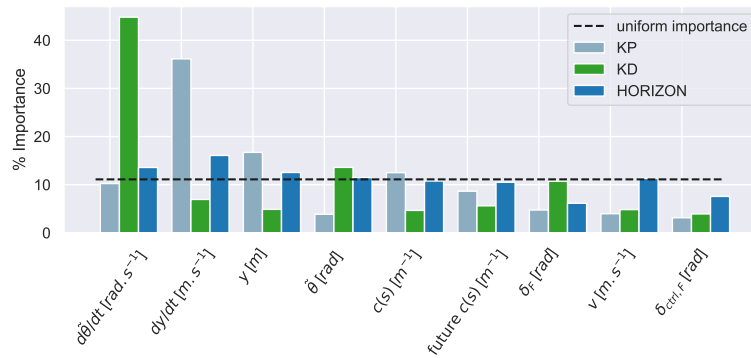


Figure 4.18: The feature importance for the *NN gain tuner* method for each input, denoted in % of importance.

From the figure 4.18, the inputs that contribute the most to the outputs of the *NN gain tuner* method are in order of importance the rate of change of the angular error denoted $d\tilde{\theta}/dt$, the rate of change of the lateral error denoted dy/dt , the lateral error denoted y , the angular error denoted $\tilde{\theta}$, the immediate curvature denoted $c(s)$, the angular error denoted θ , and the future curvature denoted future $c(s)$ which all contribute a total of at least 80% of the variations of the outputs of the *NN gain tuner* method. The control steering output denoted $\delta_{ctrl,F}$ and the speed denoted

v are significantly below the expected importance for a uniform importance distribution over the gains K_p & K_d , which implies that they are not significant for the outputted gains. However the importance of the inputs with respect to the horizon seems uniform with a noticeable exception for the steering inputs, which implies that most of the inputs are useful for predicting the horizon, which in turn demonstrates a considerable complexity for predicting the horizon.

This shows that the *NN gain tuner* is correcting the control gains output in a similar manner to the *Delta NN ctrl*. However, it is specializing its variations in a consistent manner to the expected theory of the control law, as the K_d gain varies mostly with the rate of change of the angular error and the angular error, the K_p gain varies mostly with the rate of change of the lateral error and the lateral error, and the control horizon varies with most of the inputs. It is strongly implied that the *NN gain tuner* is able to reach higher performance than *Delta NN ctrl* thanks to this specialization, which leads to simplified control outputs when compared to controlling the steering output directly. Furthermore it is likely that the tuning the gains allows for a predictable and targeted correction, which is a simpler task than predicting the control output of the existing controller in order to correct it.

Validation of the results over test trajectories

In order to validate the results shown previously, additional tests need to be run in conditions that were not present during the training phase. This is done to verify that the method has not over-fitted to the training set, and has indeed generalized to novel situations.

The following tables show the mean surface error generated by each method over multiple runs for each speed with varying grip conditions (C_R & C_F from 7000 to 30000 $N.rad^{-1}$), with GPS losses, and with testing trajectory (i.e. outside the training dataset):


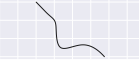

				
		estoril1_2	estoril11_12	estoril6
$1m.s^{-1}$	Romea	5.06 (± 0.81)	2.40 (± 0.54)	2.59 (± 0.45)
	EBSF	7.53 (± 0.81)	2.70 (± 0.55)	2.87 (± 0.56)
	<i>NN controller</i>	8.24 (± 0.77)	2.45 (± 0.54)	1.83 (± 0.50)
	<i>Delta NN ctrl</i>	4.77 (± 0.72)	1.55 (± 0.42)	1.87 (± 0.37)
	<i>NN gain tuner</i>	6.01 (± 0.86)	1.93 (± 0.39)	1.85 (± 0.41)
$2m.s^{-1}$	Romea	7.67 (± 1.12)	4.38 (± 1.01)	4.13 (± 0.93)
	EBSF	10.01 (± 1.76)	3.71 (± 0.94)	3.76 (± 0.96)
	<i>NN controller</i>	7.39 (± 1.09)	3.36 (± 0.82)	2.76 (± 0.82)
	<i>Delta NN ctrl</i>	6.49 (± 1.05)	3.26 (± 0.87)	2.66 (± 0.80)
	<i>NN gain tuner</i>	9.03 (± 1.65)	3.07 (± 0.66)	2.46 (± 0.80)
$3m.s^{-1}$	Romea	16.16 (± 2.01)	10.24 (± 2.06)	9.26 (± 2.00)
	EBSF	27.48 (± 6.62)	5.74 (± 1.54)	5.61 (± 1.52)
	<i>NN controller</i>	32.11 (± 5.12)	5.01 (± 0.89)	4.58 (± 1.34)
	<i>Delta NN ctrl</i>	10.71 (± 1.91)	5.64 (± 1.34)	3.79 (± 1.07)
	<i>NN gain tuner</i>	15.17 (± 4.44)	5.17 (± 1.10)	3.17 (± 1.22)
$4m.s^{-1}$	Romea	22.03 (± 4.43)	13.11 (± 2.81)	11.97 (± 3.17)
	EBSF	64.05 (± 9.72)	7.81 (± 1.93)	7.78 (± 1.62)
	<i>NN controller</i>	51.33 (± 18.73)	9.42 (± 1.97)	8.67 (± 2.24)
	<i>Delta NN ctrl</i>	26.54 (± 7.93)	10.48 (± 1.73)	6.50 (± 1.30)
	<i>NN gain tuner</i>	38.82 (± 10.67)	8.33 (± 2.23)	4.49 (± 2.28)

Table 4.9: Surface error in [m2] of each method at all the speeds used during training, over novel test trajectories, with an **initial error of 0m**

As shown in table 4.9, similar results are obtained when compared to the previous table 4.7 as the *Delta NN ctrl* can obtain good performance and is comparable with the *NN gain tuner*. However, an unusual behavior occurs over estoril1_2, which shows the *Romea* controller obtaining good results. When observing the curvature of estoril1_2 in section A.12, high oscillations can be observed, and when a low pass filtering is placed, then the performance of the *NN gain tuner*

increases dramatically. This shows an interesting limitation of the method, as it is significantly sensitive to the quality of the curvature (suspected due to a sub-sampling issue over the future curvature), and large swings can cause significant errors. However, these curvatures are unlikely to occur in general with such swings in values, which makes this issue unlikely but possible.

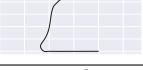
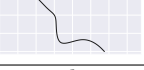

				
		estoril1_2	estoril11_12	estoril6
$1m.s^{-1}$	Romea	8.35 (± 0.74)	5.70 (± 0.53)	5.94 (± 0.42)
	EBSF	12.55 (± 0.81)	7.61 (± 0.56)	8.06 (± 0.48)
	<i>NN controller</i>	9.98 (± 0.73)	4.58 (± 0.54)	3.97 (± 0.48)
	<i>Delta NN ctrl</i>	6.05 (± 0.76)	3.36 (± 0.42)	3.69 (± 0.37)
	<i>NN gain tuner</i>	8.17 (± 0.82)	3.78 (± 0.39)	3.70 (± 0.41)
$2m.s^{-1}$	Romea	12.41 (± 1.03)	8.79 (± 1.01)	8.64 (± 0.90)
	EBSF	14.54 (± 1.80)	8.48 (± 0.92)	8.58 (± 0.96)
	<i>NN controller</i>	9.59 (± 1.05)	5.44 (± 0.81)	4.78 (± 0.79)
	<i>Delta NN ctrl</i>	8.31 (± 1.09)	4.91 (± 0.89)	4.30 (± 0.80)
	<i>NN gain tuner</i>	10.66 (± 1.56)	4.78 (± 0.67)	4.28 (± 0.77)
$3m.s^{-1}$	Romea	24.43 (± 2.06)	18.08 (± 2.00)	17.57 (± 1.93)
	EBSF	32.28 (± 6.60)	11.35 (± 1.45)	11.32 (± 1.45)
	<i>NN controller</i>	33.05 (± 4.96)	7.15 (± 0.89)	6.70 (± 1.29)
	<i>Delta NN ctrl</i>	12.38 (± 1.76)	7.34 (± 1.33)	5.44 (± 1.06)
	<i>NN gain tuner</i>	18.80 (± 5.18)	6.90 (± 1.09)	5.00 (± 1.19)
$4m.s^{-1}$	Romea	30.16 (± 4.17)	20.75 (± 2.76)	20.03 (± 3.04)
	EBSF	68.43 (± 9.58)	13.32 (± 1.84)	13.38 (± 1.58)
	<i>NN controller</i>	51.59 (± 18.13)	11.65 (± 2.02)	10.79 (± 2.16)
	<i>Delta NN ctrl</i>	28.23 (± 7.97)	12.19 (± 1.75)	8.20 (± 1.29)
	<i>NN gain tuner</i>	42.67 (± 12.10)	10.12 (± 2.28)	6.24 (± 2.25)

Table 4.10: Surface error in [m²] of each method at all the speeds used during training, over novel test trajectories, with an **initial error of 1m**

The table 4.10, shows similar results to the ones obtained when compared to the previous table 4.8 as the *NN gain tuner* is able to outperform most of the methods most of the time. However the same issue occurs as well over the estoril1_2 trajectory as shown previously.

These results show that the method is capable of obtaining comparable performance when tested over novel trajectories that are not part of the training environment.

Analysis of the approach

Overall it seems that the *NN gain tuner* is able to reach similar and in some cases better performance when compared to the *Delta NN ctrl* method for this controller, which implies this method has the highest performance as of yet, making this method a strong candidate for adapting the robot's behavior in real time.

The *NN gain tuner* method also allows for an in depth analysis of the control parameters as shown previously, which gives insight into the choices and strategies being employed by the neural network, as it is much easier to interpret control parameters when compared to interpreting control signals directly such as the steering angle.

However, the *NN gain tuner* does not assume direct control of the steering control output, this would imply sacrificing potential performance, but this was not shown here. It should be noted that, it is possible to design a controller that can demonstrate this effect, for example using a proportional steering controller, as it would not be able to easily correct any angular error, which can lead to some difficult control-ability situations when the lateral error is close to 0. As such this method must be tested for any controller in order to show that *NN gain tuner* does outperform or match the performance of the previously shown methods for any said controller.

Nonetheless, this result is quite curious as existing control parameter tuning methods do exist and are able to adapt the behavior of existing controllers without resorting to using neural networks

or machine learning. Fundamentally it would be scientifically dishonest to not compare said methods to the *NN gain tuner* method, as these methods might outperform the NN gain method. As such, the following chapter will develop and compare a deterministic control parameter method, with the previously shown *NN gain tuner* method.

Chapter 5

Gain tuning in dynamic context

The design of a deterministic model-based gain tuning method can be quite difficult, as it corresponds to the approximation of the control gain parameters, based on robot model. As such, not only a more detailed model is needed than the one used to design the controller, but also additional information is needed in order to exploit said model. As such, the deterministic control Romea exploit two observer to account for sideslip angles and varying grip conditions. The output of these observer (sideslip angles and cornering stiffness), feeding the steering control law, may also be used to determine the settling time for the robot to reach a desired yaw rate. As a result, one can deduce from these variables a settling distance for the robot to reach the trajectory, allowing to tune the gain and horizon of prediction accordingly. From these dynamic observers, the dynamic model described in section 2.3 can indeed be used in order to determine such as hereafter described.

5.1 Model-based gain tuning [*Model gain tuner*]

In this section, a novel gain tuning method for the Romea controller is derived and explained. It is achieved through the time of convergence of the robot's heading, using a dynamic model (However this approach might not be applicable in every use case, if so could consider a N-order approximation of a *NN gain tuner*, as shown in section A.4 as a deterministic substitute).

System response time

From the description of the Romea controller in section 2.5, it will be expressed that the gain K_p is correlated with a settling distance D_y for the convergence of the lateral error y ; and the gains K_p and K_d are proportional to the damping factor ξ of the control system.

Given a constant damping factor $\xi = 1$, and with a model-based upper-bound approximation of D_y , a valid approximation of the control gains can be inferred. As such, the following section describes a gain adaptation method, that uses K_p and K_d to set up the theoretical convergence distance of the robot, and adapt it with respect to the robot behavior, derived from the dynamic model.

The Romea controller described in section 2.5, imposes the following dynamics of the lateral error with respect to the curvature (as described in the equation (2.12)):

$$\frac{\partial^2 y}{\partial s^2} + K_d \frac{\partial y}{\partial s} + K_p y = 0 \quad (5.1)$$

The relation between the gains and the damping factor ξ of this second order system is the following:

$$\xi = \frac{K_d}{2\sqrt{K_p}} \quad (5.2)$$

Then, using a damping factor of $\xi = 1$, a quadratic relation between the two gains K_p and K_d is derived:

$$K_p = \frac{K_d^2}{4} \quad (5.3)$$

From those considerations, one can determine that the settling distance for a 5% tracking error can be approximated by:

$$D_y = \frac{8}{K_d} \quad (5.4)$$

As a result, one can derive the settling time for the convergence of y , according to the speed of the robot as follows:

$$T_y = \frac{8}{vK_d} \quad (5.5)$$

which describes the system's response time in closed loop, with the steering controller, pending on the gain K_d (since K_p directly relies on the derivative gain). This is a theoretical settling time imposed by the control law, which is satisfied if the robot dynamics allows such a settling time. This will depend on the robot's properties (actuator, inertia), and on the grip conditions.

Settling time for the robots yaw rate

In order to check if the settling time expected to be imposed by the controller gains is achievable in practice, let us consider the dynamic model of the mobile robot with a linear tyre model described at the Eq. (2.3). One can write:

$$\dot{X} = A(C_F, C_R)X + B(C_F, C_R)\delta_F \quad (5.6)$$

where:

$$A(C_F, C_R) = \begin{bmatrix} \frac{-L_F^2 C_F - L_R^2 C_R}{v_2 I_z} & \frac{-L_F C_F + L_R C_R}{v_2 m} \\ -\frac{L_F C_F - L_R C_R}{v_2 m} & -\frac{C_F + C_R}{v_2 m} \end{bmatrix} \quad (5.7)$$

$$B(C_F, C_R) = \begin{bmatrix} \frac{L_F C_F}{v_2 m} \\ \frac{C_F}{v_2 m} \end{bmatrix}, \quad X = \begin{bmatrix} \dot{\theta} \\ \beta \end{bmatrix}$$

By deriving Eq. (5.6) over time, and by substituting $\dot{\beta}$ with the one in Eq. (2.3), the following second order differential equation over $\omega = \dot{\theta}$ is obtained:

$$\ddot{\omega} - \mathbf{A}_{1,1}\dot{\omega} - \mathbf{A}_{1,2}\mathbf{A}_{2,1}\omega = \mathbf{A}_{1,2}\mathbf{A}_{2,2}\beta \quad (5.8)$$

The 5% response time is deduced from this second order differential equation (5.8) on the yaw rate ω . Adding the steering actuator response time τ_δ , one can derive the settling time for the yaw rate of the robot as follows:

$$T_\omega = \tau_\delta + \frac{4vI_z}{L_F^2 C_F + L_R^2 C_R} \quad (5.9)$$

Eq. (5.9) shows that C_R and C_F are inversely proportional to the settling time for the angular velocity, and that v is directly proportional to the settling time for the angular velocity. This is logical since the worse the grip conditions are, the more time it takes for a vehicle to establish a constant yaw rate with respect to a set point imposed by a constant steering angle δ_F .

Gain adaptation

In order to prevent oscillations and due to the nature of the controller, the system response time T_y is expected to be larger than the settling time T_ω . As such, using Shannon's sampling theory the condition $T_y \gg T_\omega$ has to be met. In practice, one can define the following constraint between the settling time T_y imposed by the control gain K_d and K_d to ensure the convergence of y , and the settling time T_ω required for the yaw rate to converge as:

$$T_y = \mathbf{N}T_\omega \quad (5.10)$$

with \mathbf{N} a natural number above 2 determined empirically (in the experiments, \mathbf{N} was determined as $\mathbf{N} = 4$).

To be stable, the settling time imposed by Eq. (5.5) has to be inline with robot's capabilities described in the constraint $T_y = \mathbf{N}T_\omega$. As a result, a condition for stability can be defined by introducing Eq. (5.5) to obtain a tuning gain K_d satisfying the robot's constraints:

$$K_d = \frac{8}{v\mathbf{N}T_\omega} \quad (5.11)$$

Using T_ω and K_p described at Eq. (5.9), a deterministic means of adapting the gains K_p and K_d using the velocity and cornering stiffness can be achieved.

The prediction horizon of the controller is defined as T_ω , however due to the discretization of the control loop at 10hz, the prediction horizon fluctuates very little using this method, implying an almost constant prediction horizon.

The proposed expression (5.11) relies mainly on the velocity, the grip conditions, and robot properties (inertia and actuator delays). It however does not account for the perception and observer uncertainty, which play a role in the robot's stability. This implies that the approximation of the settling distance D_y is incomplete, which implies that this method could be sub-optimal when compared to neural networks gain tuning.

Experimental setup

Control loop setup

The model-based gain tuner computes the steering control gains in real-time. These steering control gains are then passed to the controller before the steering angle is calculated. This is shown in the figure 4.13 where the *Model gain tuner* takes as inputs the speed and the front and rear cornering stiffness (denoted C_F & C_R respectively), and returns the steering control gains. The control law that is used in tandem is the Romea control law, as the model-based gain tuner was designed for said control law, and in order to preserve the comparability with the previous methods.

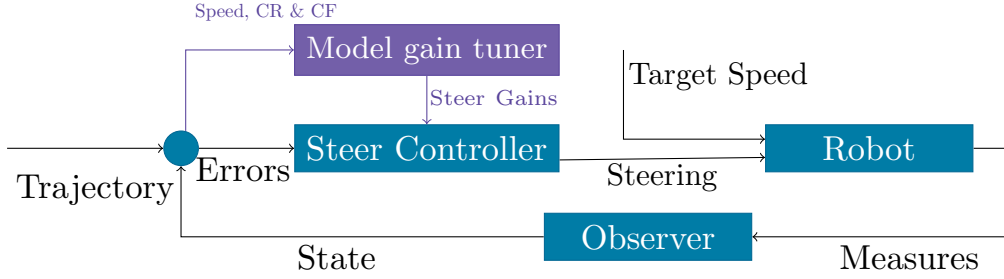


Figure 5.1: Overview of the proposed method.

The speed is defined as constant before the path following. An Extended Kalman Filter (EKF) [53] is used as part of the *Observer* in order to filter the noise from the robot's sensors, and improve the accuracy of the tracking. Following the EKF, a sliding angle observer is used [50] in order to estimate the front and rear sliding angles, which are needed to estimate the front and rear cornering stiffness detailed in section 2.7.

Metrics

The metric used for the analysis is identical to the previously defined surface error given by the equation (4.4).

$$A_{\text{error}} = \sum_{i=0}^N \left| v_i \cos(\tilde{\theta}_i) \left(y_i + \frac{v_i \sin(\tilde{\theta}_i) \Delta t}{2} \right) \right| \Delta t$$

As it will allow for minimal bias when comparing the methods.

Simulated results

Properties of the robuFAST experimental mobile platform detailed in section 5.3 are used as parameters: a wheelbase of 1.2m, 430kg of weight, a max steering angle of 15°, and a max acceleration 0.5m.s⁻² for the simulations.

The *Model gain tuner* corresponds to the Romea steering control with gains set by the model-based gain tuner introduced above. This method along with the baselines controllers, were tested in the simulation with the same speeds, trajectories, and grip conditions as used (C_F & C_R constant, sampled from 7000 to 30000 $N.rad^{-1}$) in the training of the trained method defined in

the previous sections (e.g. 4.1). These tests were run 100 times each in order to get a consistent mean and variation for each method.

The gains for the existing controllers are set to the same values defined in the section 4.1.

Quantitative Analysis

A first set of simulated runs was computed using the trajectories described in the appendix A.12. From this, the table 5.1 was obtained. It describes the Surface error from the equation (4.4), for

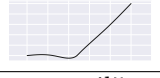

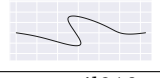

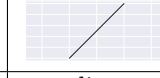
						
		estoril5	estoril7	estoril910	spline5	line
$1m.s^{-1}$	Romea	<u>1.21</u> (± 0.38)	<u>1.48</u> (± 0.75)	<u>29.15</u> (± 0.90)	<u>2.48</u> (± 1.27)	<u>1.09</u> (± 0.04)
	<i>Model gain tuner</i>	<u>1.06</u> (± 0.25)	<u>1.11</u> (± 0.57)	25.46 (± 1.15)	<u>1.98</u> (± 1.26)	<u>1.09</u> (± 0.03)
	<i>NN gain tuner</i>	<u>1.18</u> (± 0.26)	<u>1.44</u> (± 0.62)	<u>19.77</u> (± 1.95)	<u>1.99</u> (± 1.26)	<u>1.17</u> (± 0.03)
$2m.s^{-1}$	Romea	<u>1.70</u> (± 0.71)	<u>2.78</u> (± 1.22)	<u>33.95</u> (± 1.74)	<u>4.72</u> (± 1.52)	<u>1.13</u> (± 0.06)
	<i>Model gain tuner</i>	<u>1.50</u> (± 0.76)	<u>2.15</u> (± 1.36)	31.03 (± 2.65)	<u>3.67</u> (± 1.87)	<u>1.16</u> (± 0.06)
	<i>NN gain tuner</i>	<u>1.62</u> (± 0.40)	<u>2.14</u> (± 1.05)	<u>17.51</u> (± 1.82)	<u>3.19</u> (± 1.61)	<u>1.35</u> (± 0.08)
$3m.s^{-1}$	Romea	<u>3.49</u> (± 1.29)	<u>6.41</u> (± 1.75)	<u>55.00</u> (± 5.08)	<u>10.10</u> (± 1.75)	<u>1.22</u> (± 0.12)
	<i>Model gain tuner</i>	2.56 (± 1.80)	4.03 (± 2.87)	43.31 (± 8.80)	7.19 (± 3.43)	<u>1.33</u> (± 0.12)
	<i>NN gain tuner</i>	<u>2.34</u> (± 0.70)	<u>3.20</u> (± 1.42)	<u>20.61</u> (± 4.40)	<u>6.65</u> (± 3.30)	<u>1.50</u> (± 0.14)
$4m.s^{-1}$	Romea	<u>4.72</u> (± 1.62)	<u>8.60</u> (± 2.24)	59.01 (± 9.25)	12.74 (± 3.56)	<u>1.26</u> (± 0.15)
	<i>Model gain tuner</i>	4.66 (± 3.69)	7.48 (± 4.82)	<u>69.39</u> (± 23.62)	<u>14.40</u> (± 7.06)	<u>1.61</u> (± 0.25)
	<i>NN gain tuner</i>	<u>3.96</u> (± 2.27)	<u>5.38</u> (± 1.99)	<u>27.95</u> (± 6.48)	<u>12.05</u> (± 7.91)	<u>1.81</u> (± 0.24)

Table 5.1: Surface error in $[m^2]$ of each method at all the speeds and trajectories used during training, with an **initial error of 0m**.

each method at all the speeds and trajectories used during the training, with an initial error of 0m. The underlined and bold values mean that the result is significant and has a p-value below 10^{-3} , determined using the *Welch-t test* [81].

Overall, the average surface error for the previous *NN gain tuner* was noticeably lower than the proposed *Model gain tuner* method at $5.30m^2$, where as the surface error for *Romea* was $8.84m^2$ (a 40.0% reduction), and the surface error for the *Model gain tuner* method was $7.60m^2$ (a 30.3% reduction). From this table, more specific strengths and weaknesses can be observed. The *Model gain tuner* was able to match or exceed the performance of the existing controllers in most cases. However, it was not able to outmatch the performance of the *NN gain tuner* method at any speeds above $2m.s^{-1}$ with the exception of the *line* trajectory. Furthermore, it seems that the *NN gain tuner* and the *Model gain tuner* methods are quite similar in performance, when compared to the static gain *Romea* method. Overall it seems that the *NN gain tuner* method is able to outmatch the more complex *Model gain tuner* method, which requires two additional observers in order to achieve the desired performance. As with the previous methods, when adding an initial lateral error of 1m at the start of the trajectories, both the *NN gain tuner* method and the *Model gain tuner* method are capable of impressive performance, as seen in table 5.2.

Overall, the average surface error for the *NN gain tuner* was noticeably lower than the proposed *Model gain tuner* method at $7.07m^2$, where as the surface error for *Romea* was $14.7m^2$ (a 51.9% reduction), and the surface error for the *Model gain tuner* method was $9.51m^2$ (a 25.7% reduction). From this table, more specific strengths and weaknesses can be observed. The *NN gain tuner* was able to match and slightly exceed the performance of the *Model gain tuner* in most of the tested cases. In particular, both methods seem very comparable on *estoril5* and *estoril7* trajectories.

Qualitative Analysis

When focusing with a qualitative analysis over the *spline5* trajectory, at $2m.s^{-1}$ with 1m of initial error, the following results are obtained. Figure 5.2 shows the same behavior as previously described, but for the *Model gain tuner* method. Indeed the reactivity of the initial lateral error

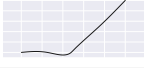



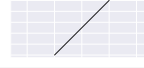
						
		estoril5	estoril7	estoril910	spline5	line
$1m.s^{-1}$	Romea	4.44 (± 0.39)	4.76 (± 0.74)	32.40 (± 0.89)	5.73 (± 1.24)	4.35 (± 0.08)
	<i>Model gain tuner</i>	2.85 (± 0.27)	2.91 (± 0.57)	27.25 (± 1.19)	3.78 (± 1.24)	2.89 (± 0.12)
	<i>NN gain tuner</i>	3.00 (± 0.26)	3.27 (± 0.61)	21.58 (± 1.96)	3.85 (± 1.23)	3.00 (± 0.08)
$2m.s^{-1}$	Romea	6.07 (± 0.71)	7.18 (± 1.22)	38.33 (± 1.74)	9.00 (± 1.51)	5.52 (± 0.10)
	<i>Model gain tuner</i>	3.31 (± 0.76)	3.99 (± 1.35)	32.83 (± 2.69)	5.49 (± 1.85)	2.99 (± 0.22)
	<i>NN gain tuner</i>	3.31 (± 0.41)	3.86 (± 1.04)	19.18 (± 1.86)	5.15 (± 1.69)	3.09 (± 0.15)
$3m.s^{-1}$	Romea	11.50 (± 1.30)	14.27 (± 1.73)	63.08 (± 5.03)	17.15 (± 1.85)	9.21 (± 0.15)
	<i>Model gain tuner</i>	4.53 (± 1.79)	6.03 (± 2.85)	45.30 (± 8.78)	9.12 (± 3.40)	3.32 (± 0.44)
	<i>NN gain tuner</i>	4.06 (± 0.73)	4.96 (± 1.43)	22.41 (± 4.23)	8.35 (± 3.40)	3.26 (± 0.28)
$4m.s^{-1}$	Romea	12.64 (± 1.68)	16.31 (± 2.20)	66.99 (± 9.24)	19.71 (± 3.60)	9.13 (± 0.17)
	<i>Model gain tuner</i>	6.67 (± 3.73)	9.53 (± 4.78)	71.38 (± 23.67)	16.43 (± 7.10)	3.66 (± 0.72)
	<i>NN gain tuner</i>	5.67 (± 2.14)	7.19 (± 2.08)	29.19 (± 5.97)	13.55 (± 8.24)	3.61 (± 0.43)

Table 5.2: Surface error in $[m^2]$ of each method at all the speeds and trajectories used during training, with an **initial error of 1m**.

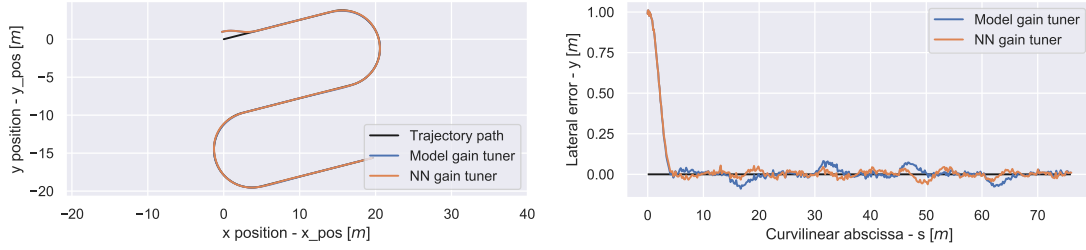


Figure 5.2: On the left: The *spline5* trajectory on a x, y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.

is very high, as the *Model gain tuner* and the *NN gain tuner* methods quickly converge to the trajectory, and are able to significantly reduce the lateral error once converged when compared to the previously described methods. However, some errors occur during the transitions between the cornering and the straight sections of the trajectory with the *Model gain tuner* method.

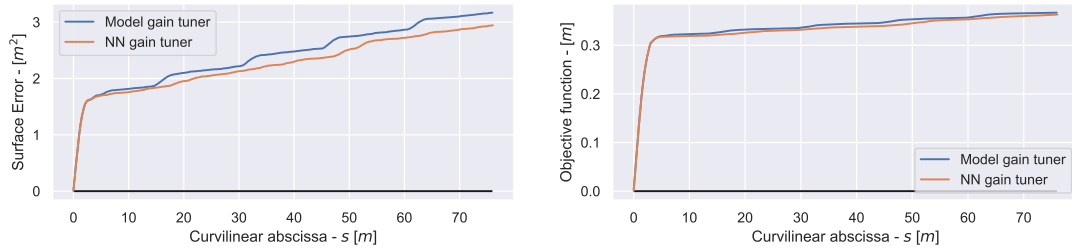


Figure 5.3: On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.

This result is quite clear on the objective function and surface error from figure 5.3, as the *Model gain tuner* is able to obtain comparable performance to the *NN gain tuner* method, with the apparent errors occurring mostly at the corners, which cause a significant cumulative difference between the methods.

The steering plots are very similar, due to the same controller being used without altering the control output. Indeed the differences are very minor and are difficult to interpret to any specific reaction for each method.

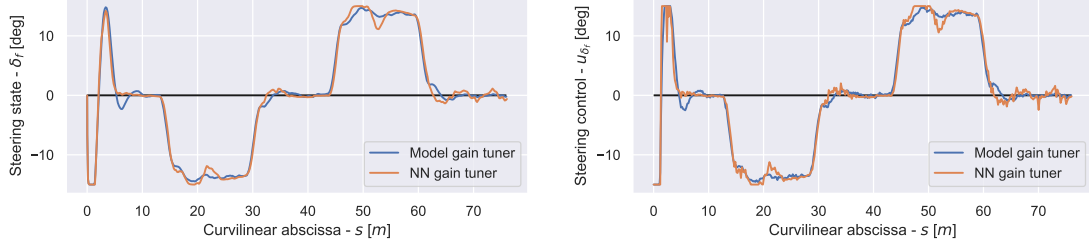


Figure 5.4: On the left: The real steering state over the curvilinear abscissa. On the right: The steering input from the controller over the curvilinear abscissa.

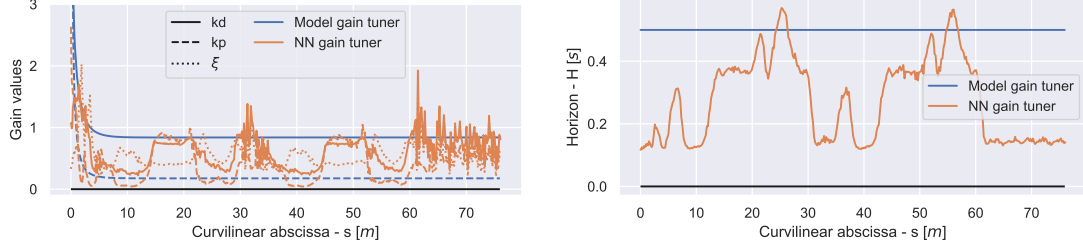


Figure 5.5: On the left: The gains over the curvilinear abscissa. On the right: The horizon over the curvilinear abscissa.

As the neural network and the model-based method are only modulating the control parameters, this means that the desired behavior is encoded in said control parameters. Figure 5.5 shows these control parameters.

The behavior of the *NN gain tuner* is identical to the previous section, as the *NN gain tuner* is compensating for the lateral error, the speed, and the cornering in real-time.

The *Model gain tuner* however has a very smooth approach, as it converges slowly as the speed increases, to a set level. This is due to the *Model gain tuner* method taking as input the speed and the cornering stiffness, which would have converged and stabilized after the initial 10m of the trajectory, leading to a very constant gain output. However, this seems to be a valid gain tuner strategy, as show the *Model gain tuner* is able to rival the *NN gain tuner*, by only using the cornering stiffness and speed, all while keeping the damping factor at $\xi = 1$, where as the NN based methods are modulating ξ implicitly through the gains.

For the horizon of the predictive section of the controller, figure 5.5 shows that the horizon is constant for the deterministic method. It is possible to correlate the horizon to the calculated system response time T_y , but in practice this leads to a horizon with a much lower performance and as such a constant value was used, in this case the constant value described in section 4.1.

An unusual behavior is also visible for the *NN gain tuner* method as the damping factor ξ is often below $\frac{\sqrt{2}}{2}$, a value below this for the damping ξ causes the control system to be considerably under-damped and would not classically be used to tune a control law due to the high oscillations that would occur. However the neural network does this in stable regions (i.e. straight lines), where a higher reactivity can allow for a fast reactivity and convergence, and then increases the damping ξ as needed in order to remain stable.

Analysis of the approach

Overall it seems that the *Model gain tuner* is able to reach similar performance when compared to the *NN gain tuner* method, which implies that even with only the speed and cornering stiffness the method was able to match the *NN gain tuner* method. This means that the *NN gain tuner* method can be used in order to reach similar performance to a deterministic gain tuning method, without needing to develop said method for every robot and controller. While the *NN gain tuner* method is also able to easily exploit inputs that would be very difficult to include into a deterministic gain tuning method.

Unfortunately, this performance similarity also seems to imply that both methods are either incomplete and that adding these parameters to the *NN gain tuner* method would improve its performance. Or it could mean that only the common input, which is the speed, is the only relevant input to the gain tuning performance.

To determine this, additional inputs are added to the *NN gain tuner* method¹. It needs to be noted that the addition of inputs element by element to the *NN gain tuner* method can be counter intuitive, as adding inputs that describe incomplete information (meaning adding a single parameter that describes the dynamics of the system) might lead to a degradation of performance.

As such, The results shown in the following section include a *set* of additional inputs, including the Kalman covariance matrix (in order to determine the validity of the observer's outputs).

5.2 Control parameter tuning using dynamic parameters [*Full NN gain tuner*]

In order to feed the controller with lateral and angular errors, the state vector has to be known. For estimating the state of the robot, an Extended Kalman filter (EKF) is used. It assists in determining the linear speed, the x, y position, and the heading. It is widely used due to its simplicity and robustness. In the control system, \mathcal{C} is the covariance matrix of the estimation and it is used to determine the level of precision of the perception. A novelty the work done is that both the estimate and the corresponding covariance matrix are integrated in the tuning of the control law. Indeed this allows the system to take into account any combination of any sensor failure in a quantifiable way, all without exposing the neural network to any redundant information, which simplifies greatly the training of said neural network.

The diagonal of the EKF covariance matrix \mathcal{C} is thus appended to the input vector of the neural network computing the control parameters. This allows the neural network to estimate the accuracy of any input with respect to the EKF covariance matrix in real time.

In this section, *NN controller* and *Delta NN ctrl* are not tested, as their performance was sub-optimal in the previous section, and this behavior was repeated even when the methods were given the same inputs as the *Full NN gain tuner* method (see appendix A.5 for details)

Experimental setup

Control loop setup

The neural network predicts the control parameters in real-time. In this case, they are the steering control gains and horizon, which are then passed to the controller before it calculates the steering angle. This is shown in the figure 5.6 where the neural network takes as inputs the errors, curvature, and speed, then returns the steering control gains and horizon. The control law that is used in tandem is the Romea control law, in order to preserve the comparability with the previous methods.

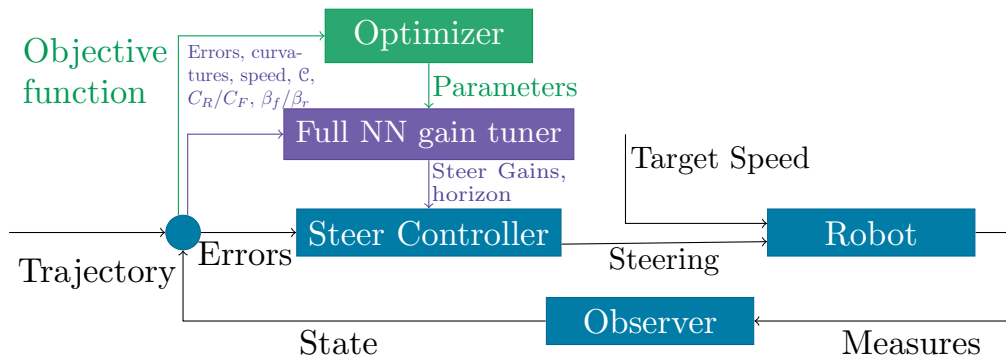


Figure 5.6: Overview of the proposed method.

The neural network is trained in a simulation using the CMA-ES optimizer depicted as the *Optimizer* in the figure 5.6, which takes as input the objective function value and returns the

¹Where adding C_R, C_F increased performance by 0.221%, $C_R, C_F, \beta_F, \beta_R$ increased performance by 1.43%, and adding $\mathcal{C}_x, \mathcal{C}_y$ increased performance by 1.62%

neural network parameters. The neural network takes as input the same information as an existing steering controller, which is the lateral error, angular error, curvature, future curvature (20 sampled points over 5s horizon), speed, and the robot's steering state. In addition it also takes as inputs the dynamic parameters, such as the cornering stiffnesses (C_R/C_F), the sliding angles (β_f/β_r), and the sensors accuracy encoded in the Kalman covariance matrix (\mathcal{C}). From these inputs, the neural network is then expected to output the control parameters. The speed is defined as constant before the path following. An Extended Kalman Filter (EKF) [53] is used as part of the *Observer* in order to filter the noise from the robot's sensors, determine the sensor accuracy, and improve the accuracy of the tracking. Following the EKF, a sliding angle observer is used [50] in order to estimate the front and rear sliding angles, which are needed to estimate the front and rear cornering stiffnesses. The latter are estimated then using a cornering stiffness observer [89].

Metrics

The objective function used is identical to the previously defined first objective function, shown in the equation (4.3).

$$obj_1 = obj_{err} + k_{steer} obj_{steer}$$

Integrating the neural network as a control parameter tuner for the steering controller does not alter the training target, since both control tasks are the same, which is to steer the robot on its path.

The metric used in the analysis is identical to the previously defined surface error shown in the equation (4.4).

$$A_{error} = \sum_{i=0}^N \left| v_i \cos(\tilde{\theta}_i) \left(y_i + \frac{v_i \sin(\tilde{\theta}_i) \Delta t}{2} \right) \right| \Delta t$$

As it will allow for minimal bias when comparing the methods.

Training details

The neural network is trained over 5 unique trajectories (*estoril5*, *estoril7*, *estoril910*, *line*, and *spline5* as shown in A.12) twice with two varying scaling factors of 1 and 2 (this is done so longer trajectories are also tested with lower curvatures), at speeds of 1.0, 2.0, 3.0, and 4.0 $m.s^{-1}$, with varying grip conditions (cornering stiffness ranging from 7000 to 30000), and with GPS losses occurring randomly across the trajectory (a 5 second signal loss, forcing a dead reckoning). Properties of the robuFAST experimental mobile platform are used as parameters: a wheelbase of 1.2m, 430kg of weight, a max steering angle of 15° , and a max acceleration $0.5m.s^{-2}$

Simulated results

The trained method along with the baselines controllers, were tested in the simulation with the same speeds, trajectories, and grip conditions (C_F & C_R constant, sampled from 7000 to 30000 $N.rad^{-1}$) as used in the training of the trained method. These tests were run 100 times each in order to get a consistent mean and variation for each method.

The gains for the existing controllers are set to the same values defined in the section 4.1.

Quantitative Analysis

A first set of simulated runs was computed using the trajectories described in the appendix A.12. From this, the table 5.3 was obtained. It describes the Surface error from the equation (4.4), for each method at all the speeds and trajectories used during the training, with an initial error of 0m. The underlined and bold values mean that the result is significant and has a p-value below 10^{-3} , determined using the *Welch-t test* [81].

Overall, the average surface error for *Full NN gain tuner* was slightly lower than the previous *NN gain tuner* method at $5.11m^2$, where as the surface error for *Romea* was $8.84m^2$ (a 42.2% reduction), the surface error for the *Model gain tuner* method was $7.60m^2$ (a 32.8% reduction), and the surface error for the *NN gain tuner* method was $5.30m^2$ (a 3.58% reduction). The *NN gain tuner* and *Full NN gain tuner* methods seem similar overall, but a difference of up to 33% can be observed in some cases. From this table, more specific strengths and weaknesses

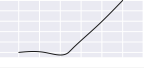



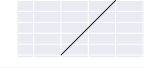
						
		estoril5	estoril7	estoril910	spline5	line
$1m.s^{-1}$	Romea	1.21 (± 0.38)	1.48 (± 0.75)	29.15 (± 0.90)	2.48 (± 1.27)	1.09 (± 0.04)
	Model gain tuner	1.06 (± 0.25)	1.11 (± 0.57)	25.46 (± 1.15)	1.98 (± 1.26)	1.09 (± 0.03)
	NN gain tuner	1.18 (± 0.26)	1.44 (± 0.62)	19.77 (± 1.95)	1.99 (± 1.26)	1.17 (± 0.03)
	Full NN gain tuner	1.14 (± 0.23)	1.23 (± 0.54)	20.94 (± 1.57)	1.64 (± 0.93)	1.17 (± 0.03)
$2m.s^{-1}$	Romea	1.70 (± 0.71)	2.78 (± 1.22)	33.95 (± 1.74)	4.72 (± 1.52)	1.13 (± 0.06)
	Model gain tuner	1.50 (± 0.76)	2.15 (± 1.36)	31.03 (± 2.65)	3.67 (± 1.87)	1.16 (± 0.06)
	NN gain tuner	1.62 (± 0.40)	2.14 (± 1.05)	17.51 (± 1.82)	3.19 (± 1.61)	1.35 (± 0.08)
	Full NN gain tuner	1.64 (± 0.41)	2.02 (± 0.99)	17.91 (± 2.09)	2.87 (± 1.20)	1.36 (± 0.07)
$3m.s^{-1}$	Romea	3.49 (± 1.29)	6.41 (± 1.75)	55.00 (± 5.08)	10.10 (± 1.75)	1.22 (± 0.12)
	Model gain tuner	2.56 (± 1.80)	4.03 (± 2.87)	43.31 (± 8.80)	7.19 (± 3.43)	1.33 (± 0.12)
	NN gain tuner	2.34 (± 0.70)	3.20 (± 1.42)	20.61 (± 4.40)	6.65 (± 3.30)	1.50 (± 0.14)
	Full NN gain tuner	2.17 (± 0.78)	3.17 (± 1.43)	18.93 (± 3.45)	5.25 (± 2.12)	1.42 (± 0.10)
$4m.s^{-1}$	Romea	4.72 (± 1.62)	8.60 (± 2.24)	59.01 (± 9.25)	12.74 (± 3.56)	1.26 (± 0.15)
	Model gain tuner	4.66 (± 3.69)	7.48 (± 4.82)	69.39 (± 23.62)	14.40 (± 7.06)	1.61 (± 0.25)
	NN gain tuner	3.96 (± 2.27)	5.38 (± 1.99)	27.95 (± 6.48)	12.05 (± 7.91)	1.81 (± 0.24)
	Full NN gain tuner	3.61 (± 2.40)	5.04 (± 2.21)	30.32 (± 54.45)	9.10 (± 3.98)	1.70 (± 0.17)

Table 5.3: Surface error in $[m^2]$ of each method at all the speeds and trajectories used during training, with an initial lateral error of $0m$.

can be observed. The **Full NN gain tuner** was able to match or exceed the performance of the previous **NN gain tuner** method in all cases. In particular, it was able to reach a slight improvement over the *estoril5* and *estoril7* trajectories, with a significant improvement over the *spline5* trajectory, and comparable performance over the *estoril910* and *line* trajectories, with similar poor behavior as shown previously for *line* trajectory when compared to the constant gain methods, due to the over-reaction of the gain tuning methods. Overall it seems that the **Full NN gain tuner** method is able to outmatch the **NN gain tuner** method in some cases, while having similar performance otherwise, which is translated in a small difference in the average surface error described previously. As with the previous methods, when adding an initial lateral error of $1m$ at the start of the trajectories, the **Full NN gain tuner** method is capable of impressive performance, as seen in table 5.4.

Overall, the average surface error for the **Full NN gain tuner** was slightly lower than the previous **NN gain tuner** method at $6.88m^2$, where as the surface error for *Romea* was $14.7m^2$ (a 53.2% reduction), the surface error for the **Model gain tuner** method was $9.51m^2$ (a 27.7% reduction), and the surface error for the **NN gain tuner** method was $7.07m^2$ (a 2.69% reduction). From this table, more specific strengths and weaknesses can be observed. The **Full NN gain tuner** method had similar performance to the **NN gain tuner** method as shown previously. This is likely due to both methods acting on the same outputs, and both integrating the lateral error, which seems to be a key input for correcting the initial error with a minimal overall error.

Qualitative Analysis

Normal conditions

When focusing with a qualitative analysis over the *spline5* trajectory, at $2m.s^{-1}$ with $1m$ of initial error, the following results are obtained. Figure 5.7 shows the same behavior as previously described. Indeed the reactivity of the initial lateral error is very high, as the **Full NN gain tuner** and the **NN gain tuner** methods quickly converge to the trajectory, and are able to significantly reduce the lateral error once converged, similarly to the **Model gain tuner** method. The difference over the lateral error between the **Full NN gain tuner** and the **NN gain tuner** methods are difficult to discern.

This results can be more accurately interpreted on the objective function and surface error from figure 5.8, as the **Full NN gain tuner** is able to obtain very comparable performance to

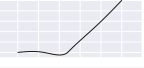




						
		estoril5	estoril7	estoril910	spline5	line
$1m.s^{-1}$	Romea	4.44 (± 0.39)	4.76 (± 0.74)	32.40 (± 0.89)	5.73 (± 1.24)	4.35 (± 0.08)
	<i>Model gain tuner</i>	2.85 (± 0.27)	2.91 (± 0.57)	27.25 (± 1.19)	3.78 (± 1.24)	2.89 (± 0.12)
	<i>NN gain tuner</i>	3.00 (± 0.26)	3.27 (± 0.61)	21.58 (± 1.96)	3.85 (± 1.23)	3.00 (± 0.08)
	<i>Full NN gain tuner</i>	2.96 (± 0.23)	3.04 (± 0.54)	22.75 (± 1.58)	3.46 (± 0.91)	2.97 (± 0.09)
$2m.s^{-1}$	Romea	6.07 (± 0.71)	7.18 (± 1.22)	38.33 (± 1.74)	9.00 (± 1.51)	5.52 (± 0.10)
	<i>Model gain tuner</i>	3.31 (± 0.76)	3.99 (± 1.35)	32.83 (± 2.69)	5.49 (± 1.85)	2.99 (± 0.22)
	<i>NN gain tuner</i>	3.31 (± 0.41)	3.86 (± 1.04)	19.18 (± 1.86)	5.15 (± 1.69)	3.09 (± 0.15)
	<i>Full NN gain tuner</i>	3.38 (± 0.41)	3.76 (± 0.99)	19.65 (± 2.11)	4.70 (± 1.29)	3.11 (± 0.14)
$3m.s^{-1}$	Romea	11.50 (± 1.30)	14.27 (± 1.73)	63.08 (± 5.03)	17.15 (± 1.85)	9.21 (± 0.15)
	<i>Model gain tuner</i>	4.53 (± 1.79)	6.03 (± 2.85)	45.30 (± 8.78)	9.12 (± 3.40)	3.32 (± 0.44)
	<i>NN gain tuner</i>	4.06 (± 0.73)	4.96 (± 1.43)	22.41 (± 4.23)	8.35 (± 3.40)	3.26 (± 0.28)
	<i>Full NN gain tuner</i>	3.96 (± 0.80)	4.98 (± 1.41)	20.76 (± 3.39)	7.33 (± 2.24)	3.27 (± 0.23)
$4m.s^{-1}$	Romea	12.64 (± 1.68)	16.31 (± 2.20)	66.99 (± 9.24)	19.71 (± 3.60)	9.13 (± 0.17)
	<i>Model gain tuner</i>	6.67 (± 3.73)	9.53 (± 4.78)	71.38 (± 23.67)	16.43 (± 7.10)	3.66 (± 0.72)
	<i>NN gain tuner</i>	5.67 (± 2.14)	7.19 (± 2.08)	29.19 (± 5.97)	13.55 (± 8.24)	3.61 (± 0.43)
	<i>Full NN gain tuner</i>	5.38 (± 2.28)	6.88 (± 2.13)	29.66 (± 12.11)	11.13 (± 3.89)	3.59 (± 0.30)

Table 5.4: Surface error in $[m^2]$ of each method at all the speeds and trajectories used during training, with an **initial error of 1m**.

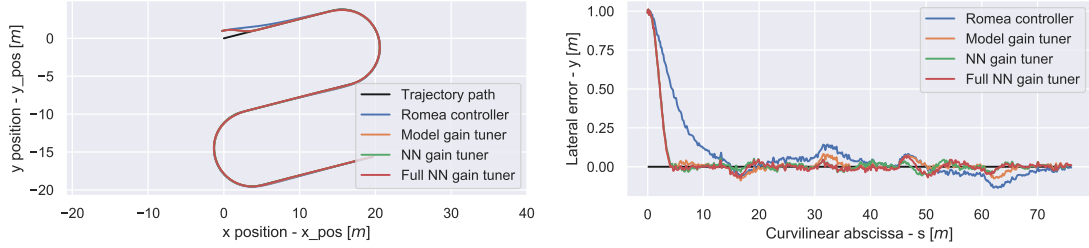


Figure 5.7: On the left: The *spline5* trajectory on a x, y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.

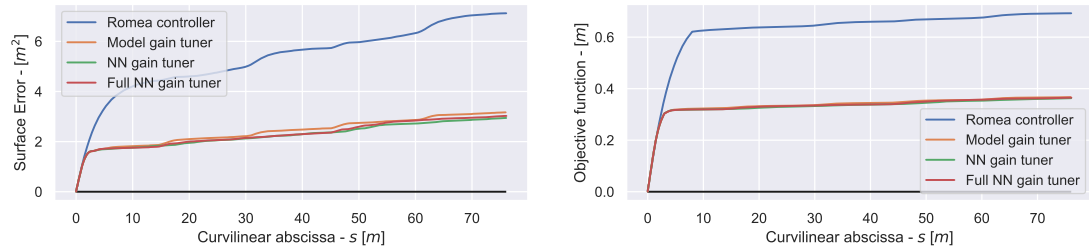


Figure 5.8: On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.

the *NN gain tuner* method.

Due to the number of outputs for each method, the damping ξ and the control gains are separated on the figure 5.9, furthermore the gains for the *Model gain tuner* are removed due to their simplicity and for readability. As the neural networks are only modulating the control parameters, this means that the desired behavior is encoded in said control parameters. However as the number of inputs increases, so does the complexity and interpretability of the output, this means that it is no obvious which inputs are influencing the gain. Due to this the feature importance becomes one of our key tools for analyzing the gains going forward.

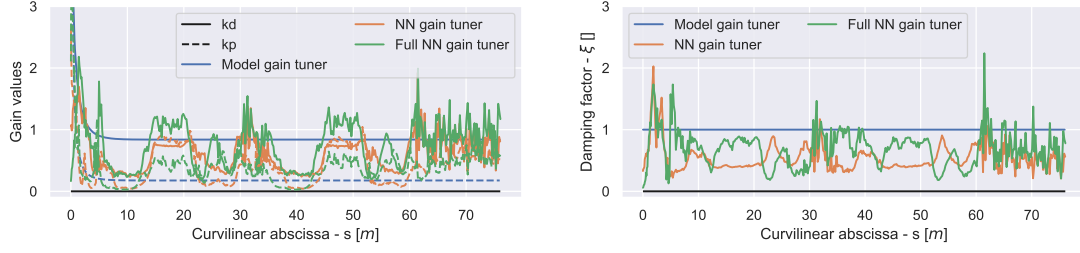


Figure 5.9: On the left: The gains over the curvilinear abscissa. On the right: The control damping over the curvilinear abscissa.

From the figure 5.9, the k_d gain is quite similar in amplitude, with the k_p gain being modulated differently. This implies a change not only with the gains, but with the damping of the control law, obtained from the equation 5.2 $\xi = \frac{K_d}{2\sqrt{K_p}}$. From this damping ξ the key differences between *NN gain tuner* and *Full NN gain tuner* can be observed, noticeably the damping over the initial sections of the corners ($15 \leq s \leq 20$ & $45 \leq s \leq 50$) is higher, which can be explained by knowing the cornering stiffness implies predictive capabilities when cornering, which lets the neural network lower the damping for smoother control in non-critical conditions. This prevents oscillations due to an under damped control, and prevents sensor noise from affecting the control output, all while keeping the robot on the desired path, due to a high enough k_d gain.

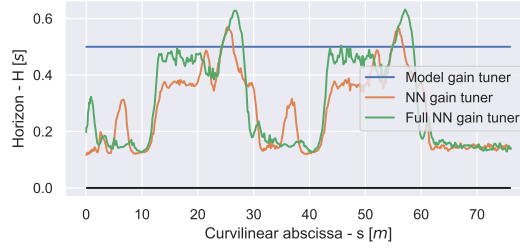


Figure 5.10: The horizon over the curvilinear abscissa.

For the horizon of the predictive section of the controller, figure 5.10 shows that the horizon is very similar for both the *NN gain tuner* and the *Full NN gain tuner* methods, with a slight increase in the corners. This can be explained as the steering response time is strongly correlated with the cornering stiffness, and as such by adding the cornering stiffness to the input, it allows the neural network to adapt the horizon as needed, depending on the grip conditions.

This analysis can be considered shallow, as the grip conditions and the sensor accuracy are not modulated, and as such the full capabilities of the *Full NN gain tuner* is not underlined, as the Kalman covariance and the cornering stiffness are effectively constant. As such the following will show results when worsening the constant grip conditions and worsening the GPS accuracy.

GPS loss

When focusing with a qualitative analysis over the *spline5* trajectory, at $2m.s^{-1}$ with $1m$ of initial error, and a GPS signal loss around the middle of the trajectory causing the system to go into dead reckoning for a short time, the following results are obtained. Figure 5.11 shows the same behavior as previously described. Except between $35 \leq s \leq 45$ where the GPS signal is lost and the system is in dead reckoning. During the dead reckoning the error seems to be low for the *NN gain tuner* and the *Model gain tuner* methods, until the GPS signal is reacquired and the estimated position snaps back to the real position. When that happens, the error becomes very significant, and drastic steering is needed in order to correct this error. This does not seem to apply to the *Full NN gain tuner*, which seems to drift on the error, but in the real position is following the trajectory very well. Once the GPS signal is reacquired the *Full NN gain tuner*

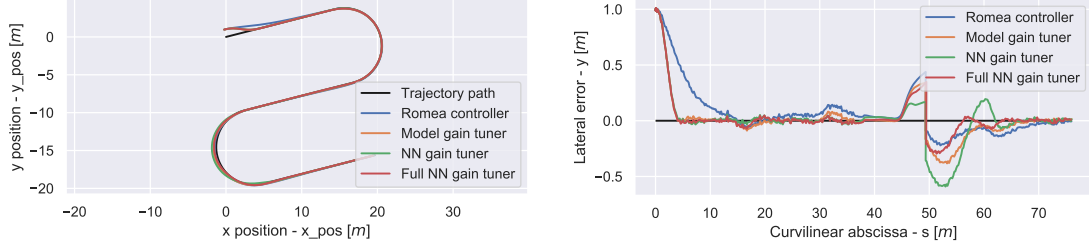


Figure 5.11: On the left: The *spline5* trajectory on a x, y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.

does not need to correct as much, as it is very close to the trajectory. An error still occurs which is dealt very well, as it has also a gap to correct even if it is lower.

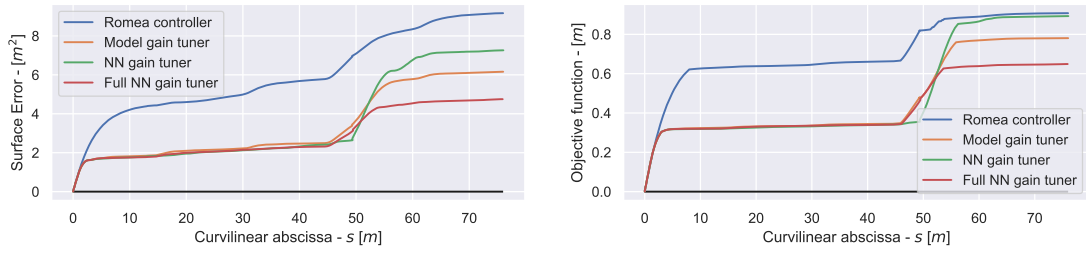


Figure 5.12: On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.

This behavior allows for the *Full NN gain tuner* method to have a much lower surface error, when compared to the other methods, as seen in the figure 5.12.

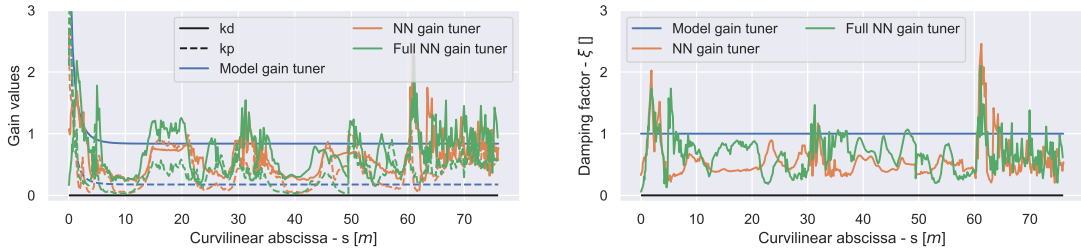


Figure 5.13: On the left: The gains over the curvilinear abscissa. On the right: The control damping over the curvilinear abscissa.

From the figure 5.13, the gains seem to drop for the *Full NN gain tuner*, along with an increase in the damping, as the Kalman covariance slowly increases due to the dead reckoning ($45 \leq s \leq 50$). This allows the controller to follow the angular error using k_d , without over-correcting the apparent error using k_p , which allows for much better path tracking.

Low grip conditions

When focusing with a qualitative analysis over the *spline5* trajectory, at $4m.s^{-1}$ with $1m$ of initial error, and low grip conditions (stiffness coefficient decreased to a constant $7000N.rad^{-1}$ value), the following results are obtained. Figure 5.14 shows the limits of the system at $4m.s^{-1}$, with low grip conditions. The errors are very high, with the lowest seems to be the *Full NN gain tuner*.

Further analysis in the figure 5.15, shows that the *Full NN gain tuner* method has a much lower surface error, when compared to the other methods, with the *Model gain tuner* method obtaining very low performance likely due to the constant damping.

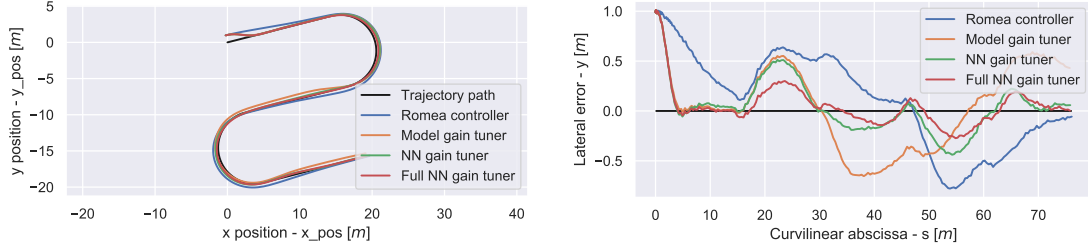


Figure 5.14: On the left: The *spline5* trajectory on a x, y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.

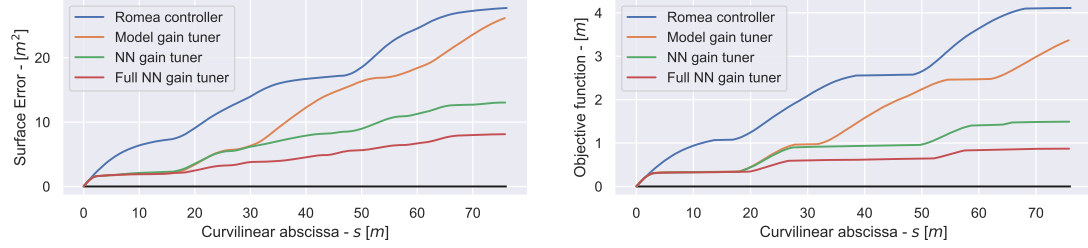


Figure 5.15: On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.

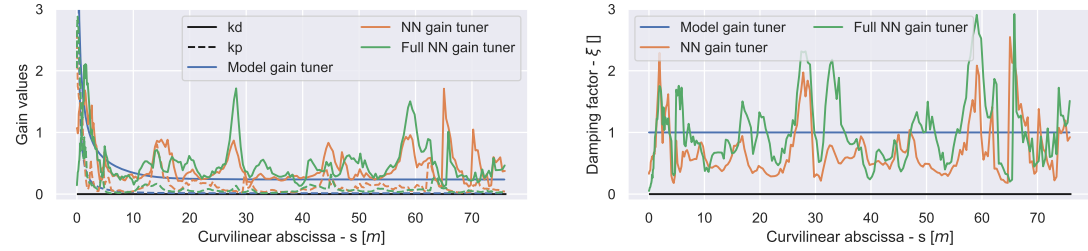


Figure 5.16: On the left: The gains over the curvilinear abscissa. On the right: The control damping over the curvilinear abscissa.

From the figure 5.16, the gains seem much higher for the *Full NN gain tuner*, along with very large damping in general and a very low damping when transitioning from and to the corner sections. This allows the controller to follow the trajectory in an over-damped mode which avoids over-steering, all while reacting correctly when critical sections occurs, which allows for much better path tracking. This underlines one of the issues with a constant damping that the *Model gain tuner* method uses. Furthermore, the *Full NN gain tuner* is more efficient than *NN gain tuner* with lower grip conditions, due to the odometry that becomes very inaccurate at low grip, which can be interpreted correctly thanks to the covariance input and cornering stiffnesses. Surprisingly, the *NN gain tuner* does not seem to determine the grip conditions, even tough it could theoretically rebuild the cornering stiffness values from its input, which implies that giving addition information that is pretreated improves the performance considerably, even tough it is entropically identical.

Feature importance

In order to better interpret and understand the neural network, a gradient based Feature importance analysis can be used to determine which inputs where useful, and quantify the utility of each input with respect to each output. See section A.4 for details on the theory and implementation of the gradient based Feature importance analysis for the neural network. Using the Feature importance analysis, the following results are obtained:

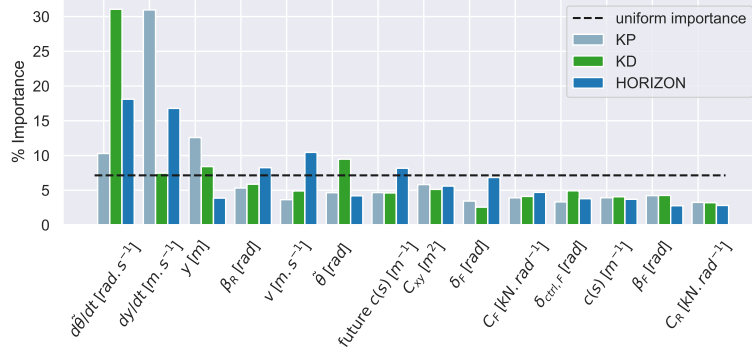


Figure 5.17: The feature importance for the *Full NN gain tuner* method for each input, denoted in % of importance.

From the figure 5.17, the inputs that contribute the most the outputs of the *Full NN gain tuner* method are in order of importance the rate of change of the angular error denoted $d\theta/dt$, the rate of change of the lateral error denoted dy/dt , which contribute a total of 40% of the variations of the outputs of the *NN gain tuner* method. The rest of the inputs seems to have a uniform importance distribution, which implies that most of the inputs are useful for predicting the outputs of the neural network, which in turn demonstrates when compared to the previous feature importance analysis that the addition of these inputs allows for a richer prediction of the control gains and horizon.

This shows that the *Full NN gain tuner* is correcting the control gains output using most of the inputs available to it. This allows for a more complex prediction of the gain and horizon, which allows a higher level of performance with respect to the objective function and metrics as shown previously. Furthermore, the neural network is able to take into account the sensor accuracy denoted \mathcal{C}_{xy} as shown previously, as it is considered more important as the cornering stiffnesses denoted C_F & C_R that is used by the *Model gain tuner* shown previously. These factors allows the *Full NN gain tuner* to outperform the *Model gain tuner* and the *NN gain tuner* when the grip conditions and the sensor accuracy change over time.

Validation of the results over test trajectories

In order to validate the results shown previously, additional tests need to be run in conditions that were not present during the training phase. This is done to verify that the method has not over-fitted to the training set, and has indeed generalized to novel situations.

The following tables show the mean surface error generated by each method over multiple runs for each speed with varying grip conditions (C_R & C_F from 7000 to 30000 $N.rad^{-1}$), with GPS losses, and with testing trajectory (i.e. outside the training dataset):

As shown in table 5.5, similar results are obtained when compared to the previous table 5.3 as the *Full NN gain tuner* obtains good performance and is comparable with the *Model gain tuner*. However, an unusual behavior occurs over estorill_2, as explained in section 4.3.

The table 5.6, shows similar results to the ones obtained when compared to the previous table 5.4 as the *Full NN gain tuner* is able to outperform each method. However the same issue occurs as well over the estorill_2 trajectory as shown previously in section 4.3.

These results show that the method is capable of obtaining comparable performance when tested over novel trajectories that are not part of the training environment.

Analysis of the approach

Overall it seems that the *Full NN gain tuner* has very good performance when compared to the *Model gain tuner* method (with good consistency in training, as shown in appendix A.6), while being able to be adaptive to sensor accuracy and grip conditions, allowing it in some case to considerably outperform the *NN gain tuner* method. This shows that the neural network only seems to be limited in the quality and quantity of input in order to obtain higher performance. Indeed, if one would add additional information, such as a pre-processed camera input, higher performance could be expected, this will be further detailed in the future works.




				
		estoril1_2	estoril11_12	estoril6
$1m.s^{-1}$	Romea	5.06 (± 0.81)	2.40 (± 0.54)	2.59 (± 0.45)
	<i>Model gain tuner</i>	4.45 (± 0.77)	1.46 (± 0.41)	1.68 (± 0.34)
	<i>NN gain tuner</i>	6.01 (± 0.86)	1.93 (± 0.39)	1.85 (± 0.41)
	<i>Full NN gain tuner</i>	4.91 (± 0.82)	1.48 (± 0.39)	1.67 (± 0.36)
$2m.s^{-1}$	Romea	7.67 (± 1.12)	4.38 (± 1.01)	4.13 (± 0.93)
	<i>Model gain tuner</i>	6.70 (± 1.30)	3.12 (± 1.18)	2.77 (± 1.14)
	<i>NN gain tuner</i>	9.03 (± 1.65)	3.07 (± 0.66)	2.46 (± 0.80)
	<i>Full NN gain tuner</i>	7.50 (± 1.02)	2.85 (± 0.74)	2.40 (± 0.75)
$3m.s^{-1}$	Romea	16.16 (± 2.01)	10.24 (± 2.06)	9.26 (± 2.00)
	<i>Model gain tuner</i>	12.72 (± 3.54)	5.87 (± 2.71)	4.26 (± 2.67)
	<i>NN gain tuner</i>	15.17 (± 4.44)	5.17 (± 1.10)	3.17 (± 1.22)
	<i>Full NN gain tuner</i>	15.47 (± 2.27)	4.83 (± 1.14)	3.12 (± 1.15)
$4m.s^{-1}$	Romea	22.03 (± 4.43)	13.11 (± 2.81)	11.97 (± 3.17)
	<i>Model gain tuner</i>	26.29 (± 11.13)	9.95 (± 5.28)	6.51 (± 4.73)
	<i>NN gain tuner</i>	38.82 (± 10.67)	8.33 (± 2.23)	4.49 (± 2.28)
	<i>Full NN gain tuner</i>	42.62 (± 10.91)	7.54 (± 2.38)	4.22 (± 2.25)

Table 5.5: Surface error in [m2] of each method at all the speeds used during training, over novel test trajectories, with an **initial error of 0m**.


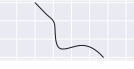

				
		estoril1_2	estoril11_12	estoril6
$1m.s^{-1}$	Romea	8.35 (± 0.74)	5.70 (± 0.53)	5.94 (± 0.42)
	<i>Model gain tuner</i>	6.31 (± 0.79)	3.30 (± 0.42)	3.49 (± 0.35)
	<i>NN gain tuner</i>	8.17 (± 0.82)	3.78 (± 0.39)	3.70 (± 0.41)
	<i>Full NN gain tuner</i>	6.87 (± 0.72)	3.30 (± 0.38)	3.47 (± 0.36)
$2m.s^{-1}$	Romea	12.41 (± 1.03)	8.79 (± 1.01)	8.64 (± 0.90)
	<i>Model gain tuner</i>	8.76 (± 1.29)	4.98 (± 1.19)	4.75 (± 1.12)
	<i>NN gain tuner</i>	10.66 (± 1.56)	4.78 (± 0.67)	4.28 (± 0.77)
	<i>Full NN gain tuner</i>	9.03 (± 1.04)	4.59 (± 0.73)	4.16 (± 0.74)
$3m.s^{-1}$	Romea	24.43 (± 2.06)	18.08 (± 2.00)	17.57 (± 1.93)
	<i>Model gain tuner</i>	15.05 (± 3.24)	7.97 (± 2.66)	6.44 (± 2.56)
	<i>NN gain tuner</i>	18.80 (± 5.18)	6.90 (± 1.09)	5.00 (± 1.19)
	<i>Full NN gain tuner</i>	17.89 (± 2.87)	6.65 (± 1.12)	5.00 (± 1.13)
$4m.s^{-1}$	Romea	30.16 (± 4.17)	20.75 (± 2.76)	20.03 (± 3.04)
	<i>Model gain tuner</i>	28.26 (± 10.97)	12.04 (± 5.24)	8.45 (± 4.70)
	<i>NN gain tuner</i>	42.67 (± 12.10)	10.12 (± 2.28)	6.24 (± 2.25)
	<i>Full NN gain tuner</i>	43.79 (± 14.89)	9.38 (± 2.39)	6.11 (± 2.28)

Table 5.6: Surface error in [m2] of each method at all the speeds used during training, over novel test trajectories, with an **initial error of 1m**.

It should be noted that the neural network is compensating for the Kalman filter's error when in dead reckoning, by altering the steering command. This implies that the control law is now dependent on the Kalman filter, which means that the neural network will need to be retrained, if the Kalman filter is changed. Furthermore, it was observed that the *Full NN gain tuner* was not able to rebuild C_R and C_F , even when given the same inputs as the cornering stiffness observer (see appendix A.8), which implies that pre-treating the information improves the performance considerably.

Unfortunately, the performance obtained seems to be the limit for the dynamic effect, as it seems that a constant speed in very low grip conditions can generate very poor path following

performance. If the speed was not constant, the controller could slow down in order to correct quickly, and then accelerate when possible. This behavior is not dissimilar to Formula-1 racing, where the acceleration control is just as important as the steering for the path following.

As such, a natural extension to this work would be to control the robot's speed in real time, while also adapting the control gains, in order to correctly control the robot at all speeds. Indeed the *Romea* control law's gains define the reactivity over time, and as such are dependent on the speed. This implies that a coupled speed and steer control system would be optimal for faster path following performance, with higher accuracy. Furthermore, with the speed control derived from the same information as the steer control, adaptive performance that depends on the sensor quality and dynamic parameter can be obtained.

The results shown in the following section explores the steering control methods previously described, along with a real time speed tuning, in order to adapt both steering and speed.

5.3 Real world experiments

In order to validate the results shown previously, experiments were run with on the RobuFAST platform, and was done for the *Model gain tuner* and the *Full NN gain tuner*.

The RobuFast robotic platform

The proposed methods have been tested on the *RobuFAST* robotic platform (Fig 5.18). The robot's mass is about 420kg, has a vertical moment of inertia of 300kg m^2 , a wheelbase of 1.2m from the center of each axles, a center of mass at 0.625m from the center of the rear axle, and a front steering response time of 0.45s. The platform runs on the ROS middleware with a control frequency of 10Hz. It has an IMU and a RTK-GPS which updates the observers and state estimators every 10Hz. The sliding angles observer is tuned for a settling time of 0.5s, and the cornering stiffness observer is tuned for a settling time of 1.5s.

The RobuFAST robot, is as robot issued from the FAST project. It has been designed as an experimental platform, that has been modified in order to reach 8.0m s^{-1} .



Figure 5.18: The RobuFAST robotic platform

Mass (m)	420Kg
Vertical moment of inertia (I_z)	300Kg m^2
Wheelbase (L)	1.2m
Front Wheelbase (LF)	0.575m
Max steering	20°
Steering action delay (Pure)	0.2s
Steering action delay (Rise time)	0.25s
Steering action delay (Total)	0.45s
Max speed	8.0m s^{-1}
Max acceleration	1.5m s^{-2}

Table 5.7: The given RobuFAST characteristics table, for reference.

Experimental Setup

We ran the experiment on the platform. Over sunny warm day and cloudless weather. With the following trajectories:

The robot starts with an initial lateral error, with a straight line to observe the stabilization from said error. A corner not too sharp to saturate the steering, but sharp enough to observe the controller's reaction to curvature. And then a straight line to observe the stabilization from the corner. The trajectory 1 starts on gravel, reaches concrete on the first half of the corner, and the rest is on dry grass. As such, it has relatively good grip, and transition on the type of ground (C_r & $C_f \sim 15000\text{N rad}^{-1}$). The trajectory 2 is on a freshly culled field of wheat, where the ground is

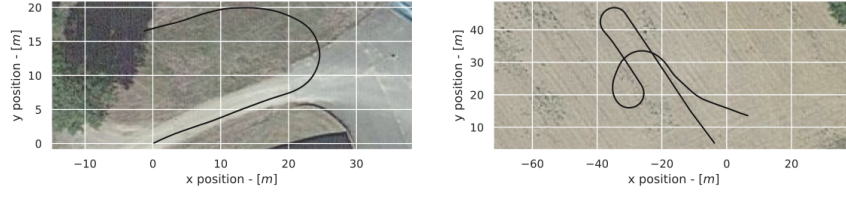


Figure 5.19: The trajectories. Left: trajectory 1. Right: trajectory 2.

uneven and covered in a layer of dust. This implies the ground had relatively poor grip conditions (C_r & $C_f \sim 10000\text{N rad}^{-1}$).

Trajectory 1

The following results were obtained at 4m.s^{-1} over the first trajectory: We can see the error over

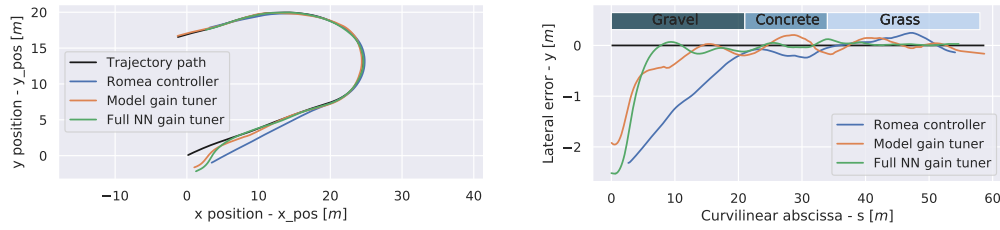
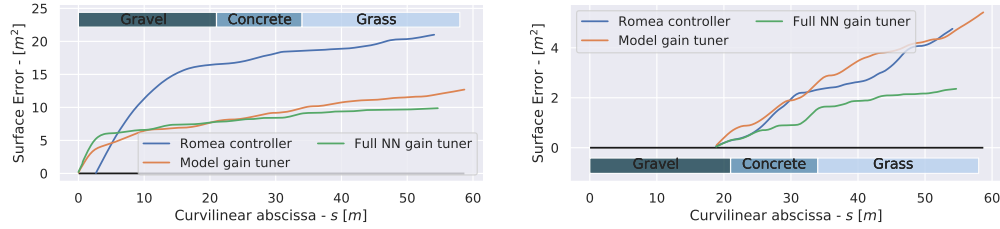


Figure 5.20: The trajectory (on the left) and the lateral error (on the right). Over the total trajectory

the entire trajectory is the lowest with the *Full NN gain tuner* method, with the *Model gain tuner* method obtaining some significant errors, and where the expert gain had the largest error.

Figure 5.21: The surface error A_{error} , and the surface error A_{error} after the initial lateral error.

A result that is reflected clearly in the surface error. Where the *Full NN gain tuner* method reaches a 62.1% reduction in the surface error, and where the *Model gain tuner* method reaches a 48.9% reduction in the surface error. If we do not include the starting error, the results are 72.3% reduction and 41.9% reduction respectively.

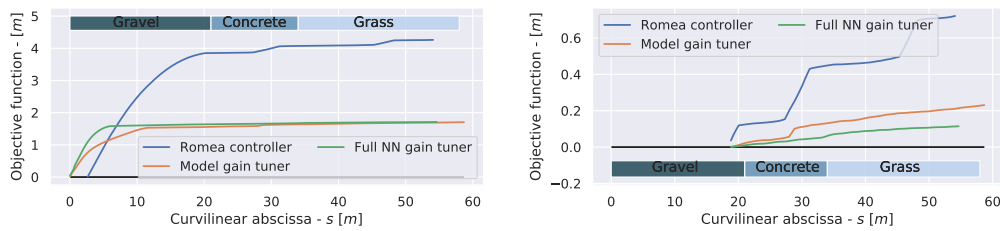


Figure 5.22: The objective function, and the objective function after the initial lateral error.

The objective function used to train the neural network, agrees with the surface error. Where the **Full NN gain tuner** method reaches a 37.5% decrease in the objective function, and where the **Model gain tuner** method reaches a 36.6% reduction in the objective function. If we do not include the starting error, the results is a 52.3% reduction and 29.8% reduction respectively.

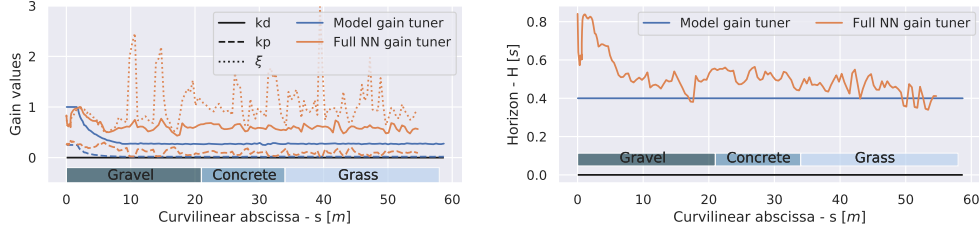


Figure 5.23: The gains, and the horizon.

The neural network gains seem to be much higher than the expert & **Model gain tuner** gains. However a strong modulation between k_p and k_d can be observed, allowing the method to dynamically update the damping factor ξ , which in turn explains the higher performance using higher gains, as a damping factor below $\xi < \frac{\sqrt{2}}{2}$ is considered unstable, but can allow for a faster convergence to the trajectory if used correctly.

Trajectory 2

The experiments with the methods were then followed up with runs at $4m.s^{-1}$ over the second trajectory: We can see, the error over the entire trajectory is significantly lower with the **Full NN**

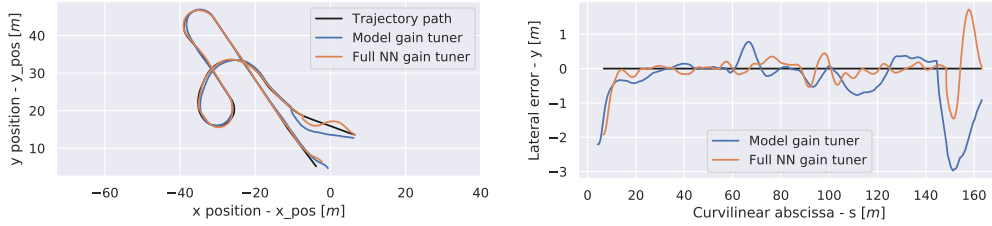


Figure 5.24: The trajectory (on the left), and the lateral error (on the right). Over the total trajectory

gain tuner method when compared to the **Model gain tuner** method.

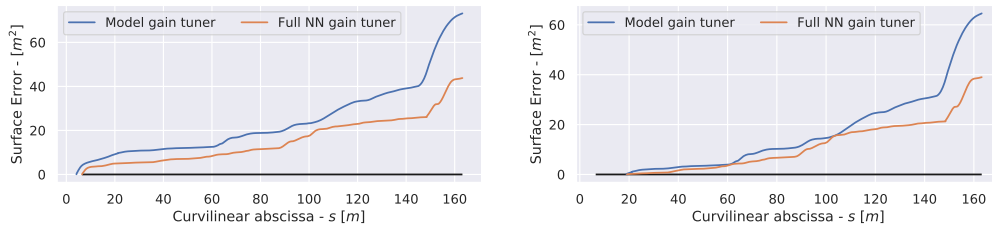


Figure 5.25: The surface error A_{error} , and the surface error A_{error} after the initial lateral error.

A result that is reflected clearly in the surface error. Where the **Full NN gain tuner** method reaches a 43.7% reduction in the surface error when compared to the **Model gain tuner** method. If we do not include the starting error, the result is a 42.9% reduction.

The objective function used to train the neural network, agrees with the surface error. Where the **Full NN gain tuner** method reaches a 31.7% decrease in the objective function when compared to the **Model gain tuner** method. If we do not include the starting error, the results is a 34.2% reduction.

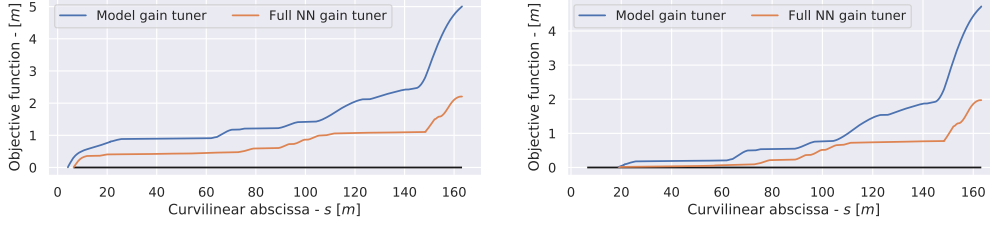


Figure 5.26: The objective function, and the objective function after the initial lateral error.

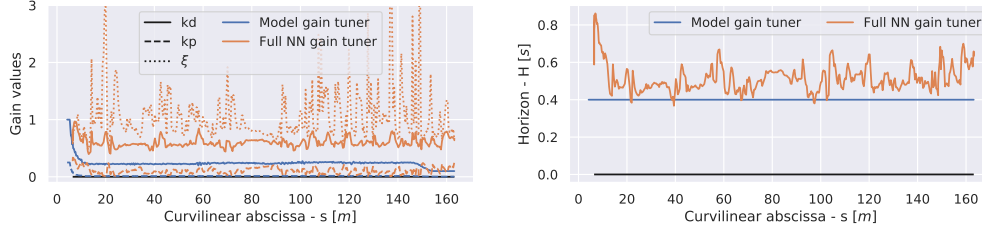


Figure 5.27: The gains, and the horizon.

As seen for the trajectory 1, the neural network gains seem to be much higher than the *Model gain tuner* gains, and an online adaption of the damping factor can be observed.

Overall, we can see that the neural network parameter tuning method, is capable of matching and even outperforming the proposed *Model gain tuner* method at $4m.s^{-1}$. However, it should be noted that the training of the neural network was unable to exceed $4m.s^{-1}$. It is assumed that this limitation is due to the task being too difficult over the training trajectories at higher speed. As such, the future experiments also modulate the maximum speed.

Conclusion

Overall, the real world experimental results show that the performance visible in simulation is transferable to reality. This is shown though the performance of the *Full NN gain tuner* which out performs considerable the constant gain and *Model gain tuner*. This is partly explained thanks to the strong modulation of the damping factor ξ , where the *Full NN gain tuner* punctually selects a $\xi < \frac{\sqrt{2}}{2}$ which is considered oscillatory, in order to converge more quickly to the desired setpoint.

These observations are supported by supplementary experimental results, provided in the appendix section A.8. This section only shown the most representative results of the numerous field experiments achieved during the PhD. Numerous experimental trials have been achieved during the PhD. The results provided in this section only shows the most representative trials, allowing an objective quantitative analysis².

Nevertheless, as can be seen from modulating the gains, the performance of the methods depend on a set gain and velocity pair. Indeed the optimal gain varies with respect to the speed, and inversely the optimal speed depends on the value of the control gains. Furthermore, there are situations where no valid gains exist for certain speeds (e.g. a velocity too high for a corner that causes a spin-out). As such, the following section considers the simultaneous speed and gain tuning, in order to improve the performance further, and allow for a greater adaptability with respect to the environment.

²As a few trials needed to be discarded due to implementation bugs.

Chapter 6

Simultaneous steer and speed control

The control parameters of a system can often be defined as a function of time or distance to convergence. As such, a given gain can be defined with an optimal speed, and inversely a given speed can be defined with an optimal gain. Naturally this means that selecting an optimal speed in real-time can be a corollary of real-time gain tuning. As such, tuning both the speed and the gains simultaneously might lead on average to lower errors with higher speeds. The following chapter will describe how to achieve such tuning for both speed and gains, while demonstrating the performance of the methods shown previously.

6.1 The problem shift due to additional speed control

When adding the speed to the steering control output, the desired task must be changed. Up until now, the task has been to follow a trajectory as accurately as possible, more formally described as following the trajectory while minimizing the lateral and angular error. This task is not longer valid when the speed and steering are both considered in tandem, as higher speeds will likely induce higher lateral and angular error. Meaning that the desired control speed for minimizing the lateral and angular error is a speed that is as low as possible. Adding speed control changes the problem of path tracking, as high speed and low error are antagonist in nature.

As such, a better suited task would be following a trajectory while authorizing a corridor of errors. This mirrors real world conditions, such as driving where small error are tolerated in order to drive at correct speeds. As such, an allowed error metric and objective function should be used to define a valid path following, instead of minimizing the overall surface error.

Methods for adjusting the longitudinal speed of the robot in real time exist, such as [90] which is based on a second order approximation on the acceleration constraints, [91, 92] which are based on models that only consider the curvature and speed limit, and [93, 94, 95] which consider the grip conditions, but are based on simplified models.

Overall, the existing works on speed tuning for complex environments with dynamic grip conditions are sub-optimal due to the use of simplified models, which do not integrate the evolution of the grip conditions over time. As such, one could consider applying a neural network output for tuning this target speed in an end-to-end fashion similarly to *NN controller*, in order to learn a control policy that is model free and is able to adapt the speed to the grip conditions accurately.

In order to integrate a speed cost penalty to the objective function, a new sub-objective function must be defined as increasing when the speed is decreased on average across the trajectory. For this, the following obj_{speed} is proposed (and derived fully later on):

$$obj_{speed} = \frac{T}{s_N}$$

Where T is the total time taken over the trajectory, s_N is the total length of the trajectory, and as such obj_{speed} describes the inverse of the average speed projected to the trajectory.

When using the previous objective function defined in equation 4.3, the following extension can be inferred:

$$obj_{1,speed} = obj_{err} + k_{steer}obj_{steer} + k_{speed}obj_{speed}$$

Or more completely:

$$obj_{1,speed} = \frac{1}{s_N} \sum_{i=0}^N [|k_{yi}y_i| + k_{steer}|Lc(s) - \tan(\delta_{Fi})|] \Delta s + k_{speed} \frac{T}{s_N}$$

However an unusual problem arises, as the different methods tested will tend to learn different behavior from one another. In practice this shows up as starkly different speeds and error compromises showing up, even with identical objective function and parameterization. This issue was determined to have originated due to the linearization of the different sub objectives in the objective functions (i.e. summing with weights), which leads to different solutions being found on the Pareto Front.

Pareto Front

In multiobjective optimization, there is a surface known as the Pareto Front [96] in the objective space. It represents the set of solutions that are attainable using the optimizer and given system. We desire to reach a high Pareto optimality, i.e. Pareto optimality is reached when improving one objective, must be degrade for an other objective.

Up until now, the Pareto Fronts have been convex (as shown in Figure 6.1), which means that the sub objective functions (i.e. obj_{err} & obj_{steer}) increase and decrease in tandem, leading to a common low error point generating a convex shape. The convexity of the sub objective functions means that when a sub objective function is minimized, the other sub objective function tends to be minimized as well, and not maximized. In our case, this shows up as when the lateral error obj_{err} is low, then the steering error obj_{steer} tends to be low as well as we are following the trajectory.

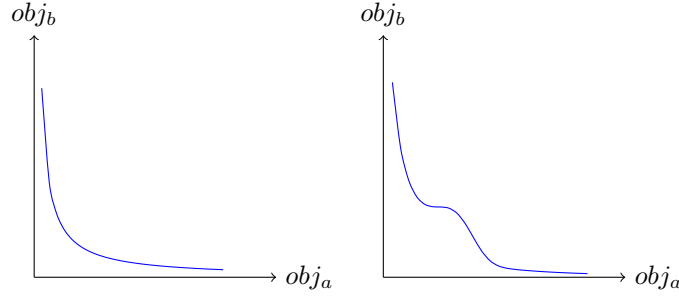


Figure 6.1: Two kinds of Pareto fronts: On the left the convex kind. On the right the non-convex kind.

When introducing a new target obj_{speed} that is minimized when the average speed is maximized, the Pareto Front is no longer non-convex (as shown in Figure 6.1). This is due to the antagonistic aspects of the speed and the errors, a higher speed almost always means a more dynamic system, and as such should be harder to control, which implies a higher error value as a compromise. Leading to two simple solutions, a simple path following at low speed (no dynamic effects), and path following that is ignored with higher speeds, leading to a complex solution that is hard to obtain between the two extremes. This means that in order to maximize the speed (i.e. minimize obj_{speed}), the system must make a trade off by increasing the error obj_{err} . This problem does not prevent training of the system and the method, but it does increase the complexity of a middle ground solution and as such the variance of the solutions found, which drastically decreases the comparability of the methods, as they are effectively optimizing different targets. One could considered that a speed objective function leading to a convex Pareto could exist, however this would imply that this function would increase the speed when the errors decrease, which would be contradictory as increasing the speed will inherently degrade tracking performance, so a speed and tracking objective seem antagonistic in nature. As such, a different linearization method must be used.

First the convex case with a weighted summing of the sub objective functions (as used up until now) is explored, in order to show why the obj_{err} & obj_{steer} did not show these issues. Then, the non-convex case with a weighted summing of the sub objective functions is used, in order to show why the obj_{err} & obj_{speed} generate different solutions when different configurations of the training

environment is used. Then finally a Weighted Hypervolume Scalarization (or Weighted Chebyshev Scalarization) is used, in order to show the advantages that this scalarization of the sub objective functions brings.

Convex case

The Pareto Fronts explored up until now have been convex in nature. When applying a weighted sum of sub objective functions $obj_{sum} = \sum_i w_i obj_i$, a hyperplane is drawn in the objective space, which is then minimized by moving the hyperplane towards the minimal points of the sub objective functions.

When different methods are used to control the robot, the Pareto Front naturally changes shape and position, due to the changes in the optimization task that were caused by allowing or disallowing some behavior as a side effect of changing control methods. Figure 6.2 shows an example of two

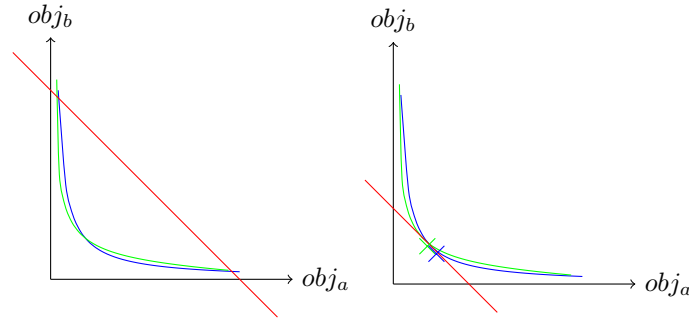


Figure 6.2: Two similar Pareto Fronts that are explored using the weighted sum scalarization. On the left: during training. On the right: Once the optimizer has converged with the crosses denoting the optimal solution found.

Pareto Fronts that are close but different, similarly to when different control methods are used. As shown, the solutions found are consistent as the minimums are close, which means that their trade offs between the sub objective functions are very similar. This explains why similar behavior is derived from the optimized solution, even when different control methods are used with the robot.

Pareto Front (non-convex case)

However, when the Pareto Fronts are non-convex in nature are used with a weighted sum of sub objective functions $obj_{sum} = \sum_i w_i obj_i$, considerably different solutions are found even with similar Pareto Fronts. Figure 6.3 shows an example of two Pareto Fronts that are close but

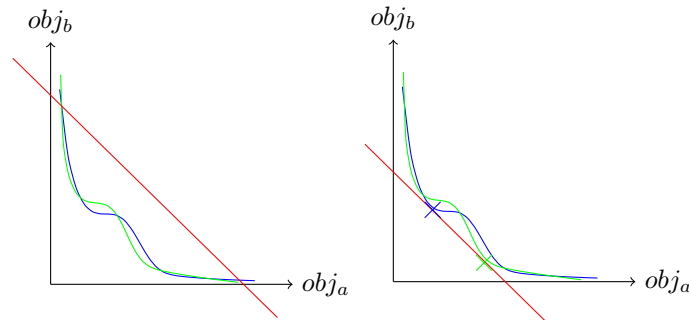


Figure 6.3: Two similar Pareto Fronts that are explored using the weighted sum scalarization. On the left: during training. On the right: Once the optimizer has converged with the crosses denoting the optimal solution found.

different, similarly to when different control methods are used. As shown, the solutions found are quite different as the minimums are not close, which means that their trade offs between the sub objective functions are different, which causes different optimization targets to be learned by the neural network. This is due to the linear nature of the scalarization which is allowed for any

solution to be equivalent, as long as the weighted sum is identical, which allows for a very high obj_{err} as long as obj_{speed} is low and vice-versa.

Pareto Front (non-convex case) with hypervolume scalarization

This problem has been studied in the literature [97, 98], and the solution to this is to use a different scalarization method. In particular the Weighted Hypervolume Scalarization (or Weighted Chebyshev Scalarization) is explored, it consists of using L_∞ normalization of the sub objective functions, which equates to taking the weighted maximum of the sub objective functions $obj_{sum} = \max_i(w_i obj_i)$. The weights assigned to the sub objective functions describe an optimization path, known as a *Utopian vector* where all the solutions along this path share the same trade offs between the objective functions, as described by the weights.

One could consider this scalarization could generate sub optimal solutions, as one of the sub objectives could be minimized but is not due to the max overriding it with a higher sub objective. However, this should not occur as this situation implies the higher sub objective can no longer be minimized but the other one can, which would imply that the sub objective functions are independent of each other near the minimum. In our case, we know that the closer the solution to the minimum, the higher the speed and the lower the error, which are not independent in dynamic environments.

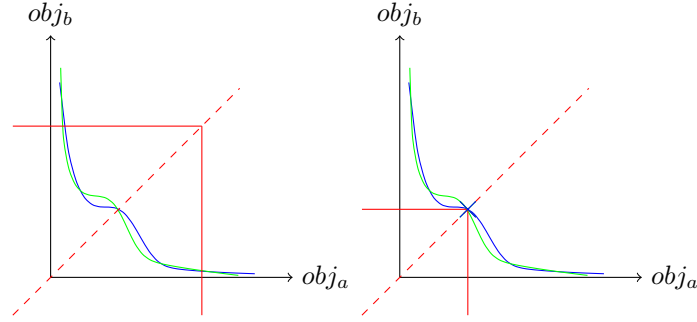


Figure 6.4: Two similar Pareto Fronts that are explored using the weighted hypervolume scalarization. On the left: during training. On the right: Once the optimizer has converged with the crosses denoting the optimal solution found.

Figure 6.4 shows an example of two Pareto Fronts that are close but different, similarly to when different control methods are used. As shown, the solutions found very close, and share the same *Utopian vector*, which means that they have the same trade off between their sub objective functions, which causes near identical optimization targets to be learned by the neural network. This allows for our methods to be comparable regardless of how the control method changes, as the average speed should increase when the average allowed error decreases.

New Objective function

Due to the limitation of linear scalarization and the allowed error described previously, a novel objective function is required.

First the allowed error needs to penalize an error that is outside of a given lateral error limit. This is already done in the high error penalty encoded in k_{yi} described in section 4.1:

$$k_{yi} = \begin{cases} k_{y \text{ low}} & \text{if } |y_i| \leq y_{lim} \\ k_{y \text{ high}} & \text{else} \end{cases}$$

and so the obj_{err} sub objective ends up identical.

$$obj_{err} = \frac{1}{s_N} \sum_{i=0}^N |k_{yi} y_i| \Delta s \quad [m] \quad (6.1)$$

The speed term needs to encode a penalty for every time step taken, normalized over the length of the trajectory. This means if the method takes more time, the speed penalty will be higher:

$$obj_{speed} = \frac{1}{s_N} \sum_{i=0}^N \Delta t = \frac{T}{s_N} \quad [\text{s m}^{-1}] \quad (6.2)$$

Interestingly, the integration of a constant penalty over time is proportional to the total time, which reduces this penalty to the inverse of the average speed taken over the whole trajectory.

For the linear scalarization, the obj_{err} and obj_{steer} are convex, as such the previous objective function obj_1 can be reused:

$$obj_1 = obj_{err} + k_{steer} obj_{steer} \quad [\text{m}] \quad (6.3)$$

However, for the speed term obj_{speed} the Hypervolume scalarization needs to be used:

$$obj_2 = \max(obj_1, k_v obj_{speed}) \quad [\text{m}] \quad (6.4)$$

Or more completely:

$$obj_2 = \max \left(\frac{1}{s_N} \sum_{i=0}^N [|k_{yi} y_i| + k_{steer} |Lc(s) - \tan(\delta_{Fi})|] \Delta s, k_v \frac{T}{s_N} \right)$$

Where k_v describes the compromise between the obj_{err} & obj_{steer} objective functions, and the obj_{speed} objective function, the higher k_v is the stronger the low speed penalty is, and as such the faster the method should be. In practice, k_v is set to 1.0 as a lower bound, and 1.25 as an upper bound¹.

A slight change is also added to the Hypervolume scalarization, where a small linear scalarization factor is added, in order to avoid issues if the Pareto Front is not convex and to improve the learning [97]. This gives us our final objective function:

$$obj_3 = obj_2 + \gamma(obj_1 + k_v obj_{speed}) \quad [\text{m}] \quad (6.5)$$

Or more completely:

$$obj_3 = \max \left(\frac{1}{s_N} \sum_{i=0}^N [|k_{yi} y_i| + k_{steer} |Lc(s) - \tan(\delta_{Fi})|] \Delta s, k_v \frac{T}{s_N} \right) + \gamma \left(\frac{1}{s_N} \sum_{i=0}^N [|k_{yi} y_i| + k_{steer} |Lc(s) - \tan(\delta_{Fi})|] \Delta s + k_v \frac{T}{s_N} \right)$$

Where γ is a small value defined as $0 < \gamma \ll 1$, usual set at $\gamma = 0.01$.

6.2 Experimental setup

Control loop setup

The neural network predicts the control parameters in real-time. In this case, they are the steering control gains, the horizon, and the target longitudinal acceleration (that is then integrated into a target speed), which are then passed to the controllers before they calculate the steering angle and wheel acceleration. This is shown in the figure 5.6 where the neural network takes as inputs the errors, curvature, and speed, then returns the steering control gains, horizon, & acceleration. The control law that is used in tandem is the Romea control law, in order to preserve the comparability with the previous methods.

The neural network is trained in a simulation using the CMA-ES optimizer depicted as the *Optimizer* in the figure 6.5, which takes as input the objective function value and returns the neural network parameters. The neural network takes as input the same information as an existing steering controller, which is the lateral error, angular error, curvature, future curvature (20 sampled

¹ k_v was determined by increasing it slowly from zero, until it reached a similar speed to the previous methods ($k_v = 1.0$), then increased until 8 m.s^{-1} was reached in simulation ($k_v = 1.25$).

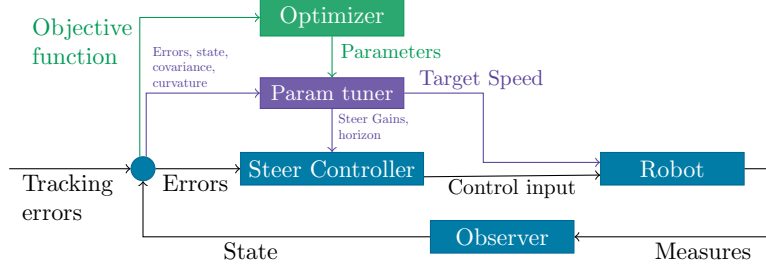


Figure 6.5: Overview of the proposed method.

points over 5s horizon), speed, and the robot's steering state. In addition it also takes as inputs the dynamic parameters, such as the cornering stiffnesses (C_R/C_F), the sliding angles (β_f/β_r), and the sensors accuracy encoded in the Kalman covariance matrix (\mathcal{C}). From these inputs, the neural network is then expected to output the control parameters and the speed setpoint. An Extended Kalman Filter (EKF) [53] is used as part of the *Observer* in order to filter the noise from the robot's sensors, determine the sensor accuracy, and improve the accuracy of the tracking. Following the EKF, a sliding angle observer is used [50] in order to estimate the front and rear sliding angles, which are needed to estimate the front and rear cornering stiffnesses. The latter are estimated then using a cornering stiffness observer [89].

Each method uses a neural network, but in different ways:

- **NN controller**: Neural network outputs the steering angle and the target speed.
- **Delta NN ctrl**: Neural network outputs a corrective term for the steering angle and the target speed.
- **Model gain tuner**: Neural network outputs only the target speed, with the steering and gains identical as described previously (section 5.1)
- **Full NN gain tuner**: Neural network outputs the K_p gain, K_d gain, and prediction horizon H for the Romea steering controller, along with the target speed.

Using this, both the speed and the steering can be modulated with respect to the external conditions, in real time. The methods are all retrained using the new objective function described above, with the new configuration.

Metrics

The objective function used is the one previously defined for this section, shown in the equation (6.5).

$$obj_3 = \max(obj_{err} + k_{steer}obj_{steer}, k_vobj_{speed}) + \gamma(obj_{err} + k_{steer}obj_{steer} + k_vobj_{speed})$$

Due to the change in task from path following to path following in a corridor, the The metric used for the analysis of the results needs to be changed. If the A_{err} is used, then the methods would be penalized unfairly for high errors within the allowed corridor. As such, we define the metric for the surface error outside the valid corridor as:

$$A_{over} = \sum_{i=0}^N \left| v_i \cos(\tilde{\theta}_i) \left(y_i + \frac{v_i \sin(\tilde{\theta}_i) \Delta t}{2} \right) \mathbb{1}_{|y_1| > y_{lim}} \right| \Delta t \quad [\text{m}^2] \quad (6.6)$$

Where $\mathbb{1}_{|y_1| > y_{lim}}$ is the indicator function that is equal to 1 when the condition $|y_1| > y_{lim}$ is met, otherwise it is equal to 0. This metric is used in order to validate the performance of the methods, without resorting to the objective function. Indeed, when a reinforcement learning agent trains to optimize a function, it is possible that the said agent might exploit the objective function in order to minimize it, without achieving the desired behavior. As such, using a different metric to measure performance allows for minimal bias when comparing the methods.

Furthermore, as we are also altering the speed we define a speed metric:

$$\bar{v} = \frac{1}{T_N} \sum_{i=0}^N |v_i| \Delta t \quad [\text{m s}^{-1}] \quad (6.7)$$

Where T_N is the total time spent over the trajectory. This metric defines the average speed over time across the trajectory.

Training details

The neural network is trained over 5 unique trajectories (*estoril5*, *estoril7*, *estoril910*, *line*, and *spline5* as shown in A.12) twice with two varying scaling factors of 1 and 2 (this is done so longer trajectories are also tested with lower curvatures), with a maximum speed of 2.0, 4.0, 6.0, and 8.0 m.s^{-1} , with varying grip conditions (cornering stiffness ranging from 7000 to 30000). Properties of the robuFAST experimental mobile platform (detailed in section 5.3) are used as parameters: a wheelbase of 1.2m, 430kg of weight, a max steering angle of 15°, and a max acceleration 0.5 m.s^{-2}

6.3 Simulated results

The trained method, were tested in the simulation with the maximum achievable speed of 8.0 m.s^{-1} , trajectories, and grip conditions as used in the training of the trained method. These tests were run 100 times each in order to get a consistent mean and variation for each method.

Quantitative Analysis

A first set of simulated runs was computed using the trajectories described in the appendix A.12. From this, the tables 6.1 and 6.2 were obtained.

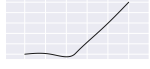

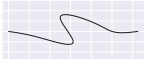


						
		estoril5	estoril7	estoril910	spline5	line
<i>NN controller</i>	A_{over}	0.52 (± 0.84)	1.11 (± 0.98)	15.44 (± 2.81)	4.46 (± 2.12)	0.34 (± 0.61)
	\bar{v}	2.22 (± 0.09)	2.29 (± 0.10)	1.82 (± 0.04)	1.39 (± 0.09)	2.45 (± 0.09)
<i>Delta NN ctrl</i>	A_{over}	13.59 (± 8.92)	7.14 (± 6.53)	91.81 (± 62.40)	3.79 (± 2.58)	16.87 (± 15.88)
	\bar{v}	3.09 (± 0.14)	2.76 (± 0.11)	3.12 (± 0.40)	1.85 (± 0.09)	3.50 (± 0.30)
<i>Model gain tuner</i>	A_{over}	0.21 (± 0.54)	1.23 (± 1.39)	19.86 (± 2.15)	2.24 (± 1.73)	0.00 (± 0.00)
	\bar{v}	3.78 (± 0.21)	3.03 (± 0.22)	2.64 (± 0.28)	2.13 (± 0.11)	5.24 (± 0.14)
<i>Full NN gain tuner</i>	A_{over}	0.32 (± 0.62)	0.88 (± 1.40)	17.05 (± 3.71)	1.69 (± 1.55)	0.00 (± 0.07)
	\bar{v}	3.75 (± 0.25)	3.05 (± 0.28)	2.72 (± 0.33)	2.11 (± 0.19)	5.18 (± 0.11)

Table 6.1: Mean speed (\bar{v}) in $[\text{m.s}^{-1}]$ and Surface error outside of the corridor (A_{over}) in $[\text{m}^2]$ for each method and trajectories used during training, with an **initial error of 0m**.

They describes the Surface error of the corridor A_{over} from the equation (6.6) and the average speed \bar{v} from the equation (6.7), for each method for all the trajectories used during the training along with the (*Big*) variants, with an initial error of 0m. The underlined and bold values mean that the result is significant and has a p-value below 10^{-3} , determined using the *Welch-t test* [81].

Overall, the *Full NN gain tuner* method had the lowest error of all with an A_{over} of 1.91 m^2 , where as the *NN controller* had an error of 2.64 m^2 (a 38.1% reduction), *Delta NN ctrl* had an error of 7.91 m^2 (a 314.0% reduction), and the *Model gain tuner* had an error of 2.42 m^2 (a 29.5% reduction). When the average speed is considered, the *Full NN gain tuner* and the *Delta NN ctrl* methods had the highest average speeds with 3.09 m.s^{-1} and 3.10 m.s^{-1} respectively (a 0.0556% difference when not rounded to 3 significant digits), where as the *NN controller* had an average speed of 2.06 m.s^{-1} (a 33.3% increase), and *Delta NN ctrl* had an average speed of 2.53 m.s^{-1} (a 18.4% increase).

Tables 6.1 and 6.2 show for the error outside of the corridor A_{over} , both the *Model gain tuner* and *Full NN gain tuner* have an overall error that is quite low and comparable between each

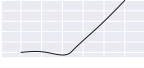



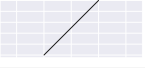
						
		estoril5	estoril7	estoril910	spline5	line
<i>NN controller</i>	A_{over}	1.64 (± 0.83)	0.87 (± 0.95)	2.47 (± 1.53)	2.35 (± 2.26)	0.33 (± 0.61)
	\bar{v}	2.43 (± 0.09)	2.57 (± 0.08)	2.39 (± 0.09)	1.85 (± 0.07)	2.45 (± 0.09)
<i>Delta NN ctrl</i>	A_{over}	23.77 (± 15.38)	10.09 (± 9.08)	15.74 (± 13.71)	10.99 (± 6.43)	16.60 (± 15.64)
	\bar{v}	3.29 (± 0.16)	3.11 (± 0.25)	3.02 (± 0.07)	2.47 (± 0.08)	3.49 (± 0.30)
<i>Model gain tuner</i>	A_{over}	0.07 (± 0.35)	0.63 (± 1.21)	1.25 (± 1.75)	2.05 (± 1.69)	0.00 (± 0.00)
	\bar{v}	4.85 (± 0.22)	3.86 (± 0.22)	4.55 (± 0.23)	2.93 (± 0.19)	5.24 (± 0.14)
<i>Full NN gain tuner</i>	A_{over}	0.41 (± 1.48)	0.41 (± 0.72)	1.50 (± 2.25)	1.32 (± 1.32)	0.01 (± 0.09)
	\bar{v}	4.93 (± 0.24)	4.01 (± 0.22)	4.59 (± 0.16)	2.90 (± 0.22)	5.18 (± 0.10)

Table 6.2: Mean speed (\bar{v} , the higher the better) in [$m.s^{-1}$] and Surface error outside of the corridor (A_{over} , the lower the better) in [m^2] for each method and the scaled trajectories used during training (Big), with an **initial error of 0m**.

other. With the *Full NN gain tuner* obtained a lower error over *estoril7* and *spline5*, but has a slight disadvantage over *estoril5*. Furthermore for the Mean speed, the *Full NN gain tuner* was able to reach a very high speed in general, with the *line* trajectory favoring slightly the *Model gain tuner*. Overall, it seems that the *Full NN gain tuner* out performed the previously described methods. Indeed it was able to reach very high speeds, while being the method with the lowest error overall.

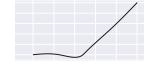

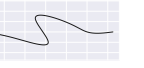
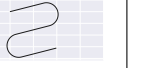
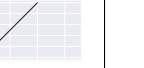
						
		estoril5	estoril7	estoril910	spline5	line
<i>NN controller</i>	A_{over}	2.62 (± 0.81)	3.25 (± 1.01)	17.53 (± 2.85)	4.39 (± 1.52)	2.46 (± 0.67)
	\bar{v}	2.21 (± 0.09)	2.27 (± 0.10)	1.81 (± 0.04)	1.36 (± 0.08)	2.43 (± 0.09)
<i>Delta NN ctrl</i>	A_{over}	2.10 (± 0.65)	3.39 (± 1.14)	16.62 (± 2.46)	3.13 (± 1.65)	2.05 (± 0.74)
	\bar{v}	2.50 (± 0.06)	2.15 (± 0.07)	2.26 (± 0.07)	1.60 (± 0.07)	2.85 (± 0.08)
<i>Model gain tuner</i>	A_{over}	1.82 (± 0.49)	2.18 (± 0.92)	20.47 (± 1.26)	2.87 (± 1.32)	1.63 (± 0.15)
	\bar{v}	3.68 (± 0.21)	2.94 (± 0.21)	2.54 (± 0.22)	1.91 (± 0.08)	5.16 (± 0.18)
<i>Full NN gain tuner</i>	A_{over}	1.92 (± 0.41)	2.27 (± 0.99)	15.84 (± 3.42)	2.66 (± 1.20)	1.63 (± 0.09)
	\bar{v}	3.66 (± 0.23)	2.95 (± 0.25)	2.50 (± 0.32)	2.03 (± 0.16)	5.06 (± 0.17)

Table 6.3: Mean speed (\bar{v} , the higher the better) in [$m.s^{-1}$] and Surface error outside of the corridor (A_{over} , the lower the better) in [m^2] for each method and trajectories used during training, with an **initial error of 1m**.

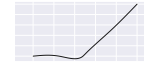
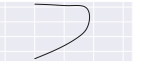
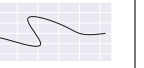

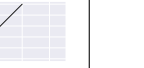
						
		estoril5	estoril7	estoril910	spline5	line
<i>NN controller</i>	A_{over}	3.83 (± 0.85)	2.98 (± 0.96)	4.58 (± 1.50)	4.45 (± 2.23)	2.44 (± 0.63)
	\bar{v}	2.42 (± 0.09)	2.55 (± 0.08)	2.38 (± 0.09)	1.84 (± 0.07)	2.43 (± 0.09)
<i>Delta NN ctrl</i>	A_{over}	2.13 (± 0.64)	2.77 (± 0.81)	3.67 (± 1.17)	2.71 (± 1.00)	2.05 (± 0.75)
	\bar{v}	2.88 (± 0.07)	2.52 (± 0.03)	2.83 (± 0.07)	2.07 (± 0.04)	2.85 (± 0.08)
<i>Model gain tuner</i>	A_{over}	1.73 (± 0.17)	1.87 (± 0.60)	2.83 (± 1.75)	2.77 (± 1.31)	1.63 (± 0.13)
	\bar{v}	4.79 (± 0.20)	3.78 (± 0.22)	4.53 (± 0.23)	2.82 (± 0.18)	5.16 (± 0.18)
<i>Full NN gain tuner</i>	A_{over}	1.94 (± 0.62)	1.94 (± 0.53)	2.86 (± 2.12)	2.66 (± 1.22)	1.63 (± 0.09)
	\bar{v}	4.84 (± 0.19)	3.95 (± 0.21)	4.58 (± 0.16)	2.85 (± 0.20)	5.07 (± 0.17)

Table 6.4: Mean speed (\bar{v} , the higher the better) in [$m.s^{-1}$] and Surface error outside of the corridor (A_{over} , the lower the better) in [m^2] for each method and the scaled trajectories used during training (Big), with an **initial error of 1m**.

For the tables 6.3 and 6.4, the error seems similar between the *Model gain tuner* and *Full NN gain tuner* with a slight advantage towards *Model gain tuner* which is able to outperform slightly the *Full NN gain tuner* for the error over the *estoril5* and *estoril7* trajectories.

With an initial error of $1m$, the performance shifts slightly. Overall, the *Full NN gain tuner* method had the lowest error of all with an A_{over} of $3.41m^2$, where as the *NN controller* had an error of $4.67m^2$ (a 37.0% reduction), *Delta NN ctrl* had an error of $3.97m^2$ (a 16.5% reduction), and the *Model gain tuner* had an error of $3.89m^2$ (a 14.2% reduction). When the average speed is considered, the *Full NN gain tuner* and the *Delta NN ctrl* methods had the highest average speeds with $3.05m.s^{-1}$ and $3.05m.s^{-1}$ respectively (a 0.155% difference when not rounded to 3 significant digits), where as the *NN controller* had an average speed of $2.05m.s^{-1}$ (a 32.8% increase), and *Delta NN ctrl* had an average speed of $2.29m.s^{-1}$ (a 25.2% increase).

Tables 6.3 and 6.4 show for the error outside of the corridor A_{over} , both the *Model gain tuner* and *Full NN gain tuner* have an overall error that is quite low and comparable between each other. With the *Full NN gain tuner* obtained a lower error over *estoril910* and *spline5*, but has a slight disadvantage over *estoril5* and *estoril7*. Furthermore for the Mean speed, the *Full NN gain tuner* was able to reach a very high speed in general, with the *line* trajectory favoring slightly the *Model gain tuner*. Overall, it seems that the *Full NN gain tuner* out performed the previously described methods as with the $1m$ initial error. Indeed it was able to reach very high speeds, while being the method with the lowest error overall. These results seems to be a repeat of the previous ones, which is expected as the neural network that tunes the speed will always slow down to minimize the surface error outside the corridor. As such, comparable results are obtained regardless of the initial error.

Qualitative Analysis

Normal conditions

When focusing with a qualitative analysis over the *spline5* trajectory, with $1m$ of initial error and a maximal speed of $8m.s^{-1}$, the following results are obtained. Figure 6.6 shows all the methods

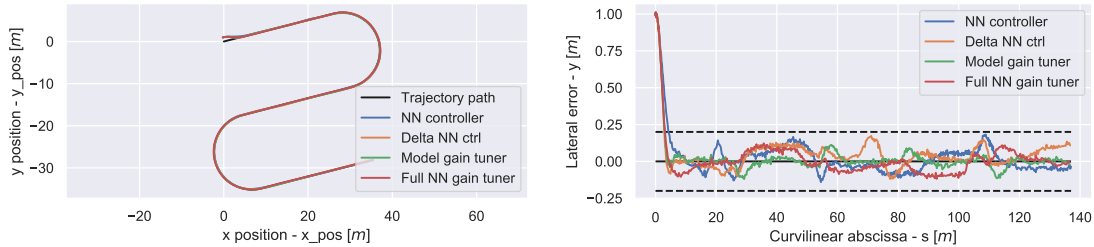


Figure 6.6: On the left: The *spline5* trajectory on a x, y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.

over the trajectory and over the error corridor. Over the trajectory all the methods stay within the corridor of allowed errors, which implies they are comparable in performance.

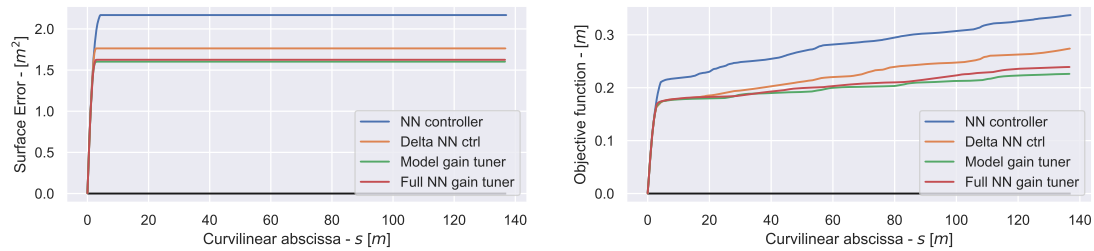


Figure 6.7: On the left: The surface corridor error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.

A result that is consistent over the surface corridor error and the objective function figure 6.7, as the *Full NN gain tuner* and the *Model gain tuner* seems to obtain very similar performances with minimal errors.

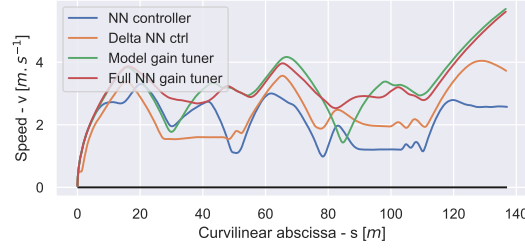


Figure 6.8: The speed over the curvilinear abscissa.

A slight difference can be observed over the speed shown on figure 6.8, where the *Full NN gain tuner* has an average speed of $3.01m.s^{-1}$, and the *Model gain tuner* has an average speed of $2.89m.s^{-1}$ (a 4.08% difference). However, this difference is rather small, and can be due to the variations in the path tracking. For the *NN controller* and the *Delta NN ctrl*, the speeds are considerably lower than the *Full NN gain tuner* and the *Model gain tuner* methods, which implies that they were not able to reach a higher speed threshold while staying within the allowed corridor.

GPS loss

When focusing with a qualitative analysis over the *spline5* trajectory, with $1m$ of initial error, a maximal speed of $8m.s^{-1}$, and with a GPS loss occurring mid trajectory, the following results are obtained. Figure 6.9 shows all the methods over the trajectory and over the error corridor. Over

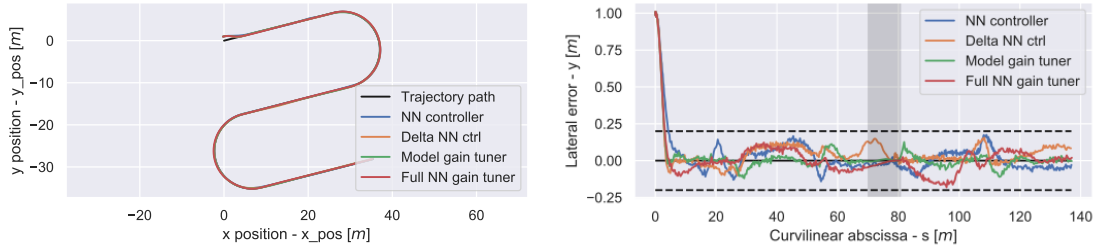


Figure 6.9: On the left: The *spline5* trajectory on a x, y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa (GPS loss zone in gray).

the trajectory the results seems almost identical to the previous results.

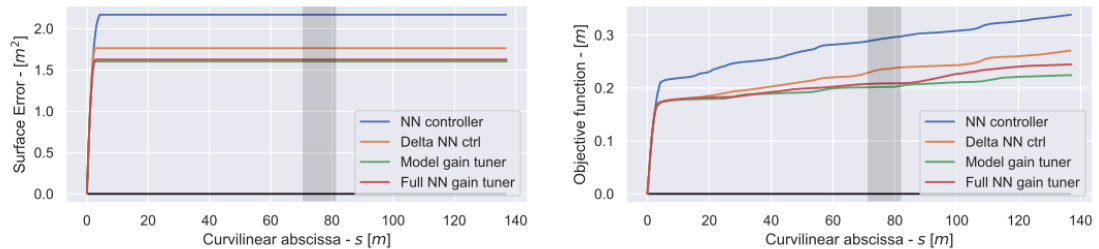


Figure 6.10: On the left: The surface corridor error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa (GPS loss zone in gray).

A result also observed over the Figure 6.10 which seems almost identical to the previous results.

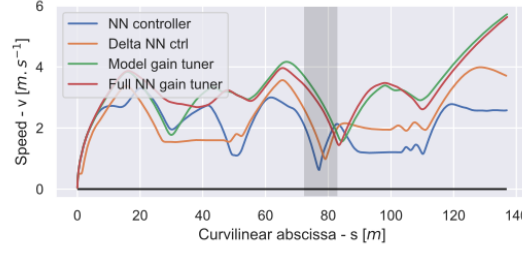


Figure 6.11: The speed over the curvilinear abscissa (GPS loss zone in gray).

When the speed is considered, no significant difference can be observed between the average speed of the *Full NN gain tuner* and the *Model gain tuner* methods. This seems to contradict the results shown in the previous chapter, however this is likely due to the speed being reduced when the GPS noise occurs at $s = 70$, allowing the *Model gain tuner* to minimize the error with respect to the sensor accuracy degradation, similarly as the *Full NN gain tuner* was achieving over tuning the control parameters alone.

Feature importance

In order to better interpret and understand the neural network, a gradient based Feature importance analysis can be used to determine which inputs were useful, and quantify the utility of each input with respect to each output. See section A.4 for details on the theory and implementation of the gradient based Feature importance analysis for the neural network. Using the Feature importance analysis, the following results are obtained:

NN controller

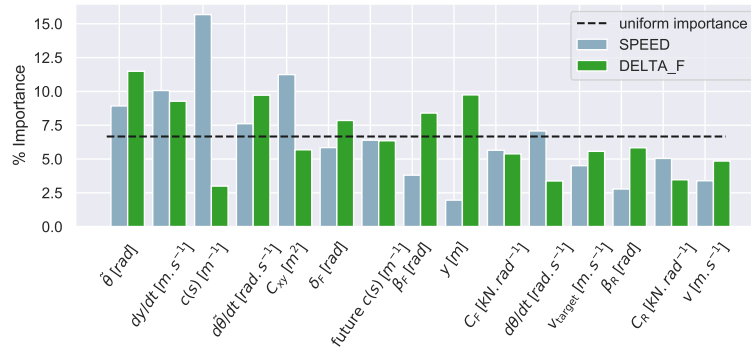


Figure 6.12: The feature importance for the *NN controller* method for each input, denoted in % of importance.

From the figure 6.12, the inputs that contribute the most the steering output of the *NN controller* method are in order of importance the angular error denoted $\tilde{\theta}$, the lateral error denoted y , the rate of change of the angular error denoted $d\tilde{\theta}/dt$, the rate of change of the lateral error denoted dy/dt , the front sliding angle denoted β_F , the front steering state denoted δ_F , and the future curvature denoted future $c(s)$, which contribute a total of 50% of the variations of the steering output. For the speed output, the inputs that contribute the most in order of importance are the immediate curvature denoted $c(s)$, the sensor accuracy denoted \mathcal{C}_{xy} , the rate of change of the lateral error denoted dy/dt , the angular error denoted $\tilde{\theta}$, the rate of change of the angular error denoted $d\tilde{\theta}/dt$, and the angular velocity denoted $d\theta/dt$, which contribute a total of 60% of the variations of the speed output. The rest of the inputs seems to have a uniform importance distribution, which implies that most of the inputs are useful for predicting the outputs of the neural network.

This shows that the *NN controller* is using the same inputs as previously shown, however is has also included the sliding angles into its inputs for the steering outputs, and the speed seems

mostly dependent on the trajectory, the sensor accuracy, the tracking errors, and the angular velocity. This means that the method is trying to avoid significant errors, while avoiding strong angular error which would be indicative of dynamic effects, and slowing down when the GPS noise is high.

Delta NN ctrl

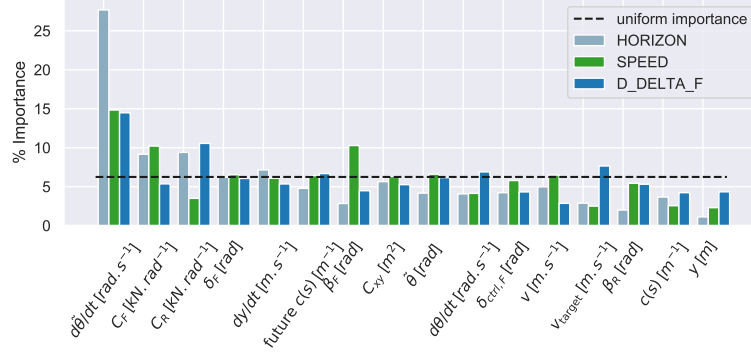


Figure 6.13: The feature importance for the *Delta NN ctrl* method for each input, denoted in % of importance.

From the figure 6.13, the inputs that contribute the most the horizon output of the *Delta NN ctrl* method are in order of importance the rate of change of the angular error denoted $d\hat{\theta}/dt$, the rear cornering stiffness denoted C_R , the front cornering stiffness denoted C_F , the rate of change of the lateral error denoted dy/dt , the front steering state denoted δ_F , the sensor accuracy denoted C_{xy} , and the future curvature denoted future $c(s)$, which contribute a total of 69% of the variations of the horizon output. For the speed output, the inputs that contribute the most in order of importance are the rate of change of the angular error denoted $d\hat{\theta}/dt$, the front sliding angles β_F , and the front cornering stiffness denoted C_F , which contribute a total of 35% of the variations of the speed output. The rest of the inputs seems to have a uniform importance distribution, which implies that most of the inputs are useful for predicting the outputs of the neural network.

This shows that the *Delta NN ctrl* is correcting the steering and horizon using most of the inputs available to it. Along with the speed it seems that the method is taking into account most of the parameters to avoid significant errors, while having a speed tuning that depends mostly on the dynamic parameters and the rate of change of the errors.

Model gain tuner

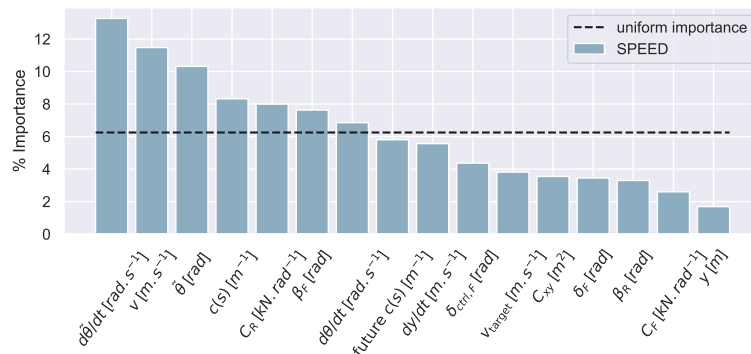


Figure 6.14: The feature importance for the *Model gain tuner* method for each input, denoted in % of importance.

For this case, only the speed is controlled using a neural network, as such for the feature importance only the speed is analyzed. From the figure 6.14, the inputs that contribute the most the speed of the *Model gain tuner* method are in order of importance the rate of change of the

angular error denoted $d\tilde{\theta}/dt$, the speed of the robot denoted v , the angular error denoted $\tilde{\theta}$, the immediate curvature denoted $c(s)$, the rear cornering stiffness denoted C_R , and the front sliding angle denoted β_F , which contribute a total of 59% of the variations of the speed. The rest of the inputs seems to have slowly decreasing but on negligible importance, which implies that most of the inputs are useful for predicting the speed, with a noticeable exception for the lateral error denoted y .

This shows that the speed tuning for the *Model gain tuner* method is basing the speed prediction mostly on the angular error, the robot's speed, the dynamic grip parameters, the angular velocity, and the trajectory. This allows for a speed that takes into account not only the trajectory's shape, but also the current state of the robot's errors, and what is allowable dynamically due to the grip conditions. This allows for a very complex and rich speed prediction.

Full NN gain tuner

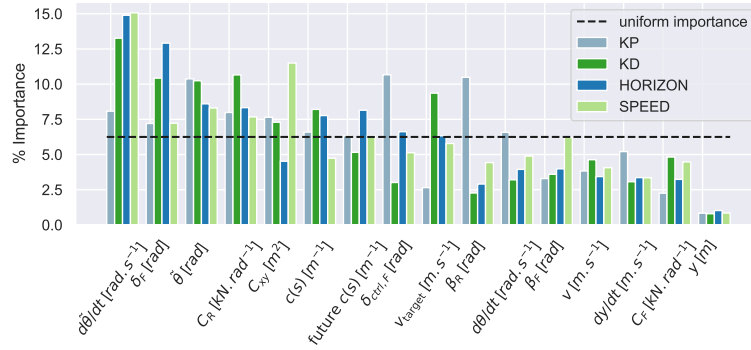


Figure 6.15: The feature importance for the *Full NN gain tuner* method for each input, denoted in % of importance.

For this strategy, many parameters are modulated by the neural network, specifically the K_p & K_d gains, the control prediction horizon, and the target speed. As such, the feature importance is done over those parameters and analyzed. From the figure 6.15, most of the inputs contribute to the overall outputs of the *Full NN gain tuner* method, with the exception of the lateral error denoted y . For each output however some outliers stand out. For the speed, the most important inputs are in order the rate of change of the angular error denoted $d\tilde{\theta}/dt$, the sensor accuracy denoted C_{xy} , the angular error denoted $\tilde{\theta}$, and the rear cornering stiffness denoted C_R . For the K_p gain, the most important inputs are in order the steer control denoted $\delta_{ctrl,F}$, the rear sliding angle denoted β_R , the angular error denoted $\tilde{\theta}$, the rear cornering stiffness denoted C_R , the rate of change of the angular error denoted $d\tilde{\theta}/dt$, and the sensor accuracy denoted C_{xy} . For the K_d gain, the most important inputs are in order the rate of change of the angular error denoted $d\tilde{\theta}/dt$, the rear cornering stiffness denoted C_R , the steering state denoted δ_F , the angular error denoted $\tilde{\theta}$, and the target speed denoted v_{target} .

This shows that the *Full NN gain tuner* is correcting the control gains and speed using most of the inputs available to it. This allows for a more complex prediction of the speed, gain, and horizon. Furthermore, the neural network is able to specialize the inputs for each output, such as the steering control and sliding angle for the K_p gain, which can be used as a good indicator of future lateral errors. The angular error, the sensor accuracy, and the cornering stiffness is then used for the speed control, allow it to depend on not only the grip conditions, but also the sensor quality and the state of the robot's tracking performance. And finally the speed, the angular error, steering state, cornering stiffness, and curvature are used for the K_d gain, allowing it to adapt to the trajectory, the speed, and the tracking performance of the robot. All these allows for a nuanced and rich adaptation of the control law in real time, using all the information available in the control loop.

Validation of the results over test trajectories

In order to validate the results shown previously, additional tests need to be run in conditions that were not present during the training phase. This is done to verify that the method has not

over-fitted to the training set, and has indeed generalized to novel situations.

The following tables show the mean surface error generated by each method over multiple runs for each speed and trajectory:

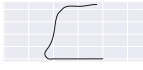


				
		estoril1_2	estoril11_12	estoril6
<i>NN controller</i>	A_{over}	15.01 (± 4.37)	1.97 (± 1.18)	2.89 (± 1.32)
	\bar{v}	2.20 (± 0.12)	1.99 (± 0.08)	2.02 (± 0.13)
<i>Delta NN ctrl</i>	A_{over}	48.93 (± 20.03)	9.42 (± 5.90)	28.05 (± 19.87)
	\bar{v}	2.77 (± 0.12)	2.45 (± 0.06)	2.94 (± 0.17)
<i>Model gain tuner</i>	A_{over}	6.84 (± 2.93)	0.78 (± 1.05)	0.63 (± 1.06)
	\bar{v}	2.66 (± 0.19)	2.69 (± 0.14)	3.67 (± 0.26)
<i>Full NN gain tuner</i>	A_{over}	9.37 (± 3.68)	0.88 (± 0.79)	0.77 (± 1.16)
	\bar{v}	2.64 (± 0.17)	2.70 (± 0.17)	3.77 (± 0.27)

Table 6.5: Mean speed (\bar{v} , the higher the better) in [$m.s^{-1}$] and Surface error outside of the corridor (A_{over} , the lower the better) in [m^2] for each method, over novel test trajectories, with an **initial error of 0m**.

As shown in table 5.5, similar results are obtained when compared to the previous table 6.1 as the *Full NN gain tuner* and *Model gain tuner* both obtain comparable performances. However, an unusual behavior occurs over estoril1_2, as explained in section 4.3.



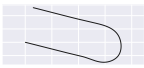
				
		estoril1_2	estoril11_12	estoril6
<i>NN controller</i>	A_{over}	17.19 (± 4.42)	3.70 (± 1.10)	5.02 (± 1.30)
	\bar{v}	2.19 (± 0.13)	1.97 (± 0.08)	2.01 (± 0.13)
<i>Delta NN ctrl</i>	A_{over}	17.42 (± 6.30)	4.17 (± 1.56)	2.11 (± 0.51)
	\bar{v}	2.09 (± 0.14)	2.09 (± 0.06)	2.45 (± 0.08)
<i>Model gain tuner</i>	A_{over}	6.47 (± 2.28)	2.20 (± 0.86)	1.96 (± 0.53)
	\bar{v}	2.55 (± 0.16)	2.60 (± 0.14)	3.62 (± 0.23)
<i>Full NN gain tuner</i>	A_{over}	8.47 (± 2.46)	2.43 (± 0.68)	2.31 (± 1.11)
	\bar{v}	2.37 (± 0.22)	2.63 (± 0.15)	3.69 (± 0.26)

Table 6.6: Mean speed (\bar{v} , the higher the better) in [$m.s^{-1}$] and Surface error outside of the corridor (A_{over} , the lower the better) in [m^2] for each method, over novel test trajectories, with an **initial error of 1m**.

The table 5.6, shows similar results to the ones obtained when compared to the previous table 6.3 as the *Full NN gain tuner* and *Model gain tuner* both obtain comparable performances, while outperforming significantly the end-to-end *NN controller* approach. However the same issue occurs as well over the estoril1_2 trajectory as shown previously in section 4.3.

These results show that the method is capable of obtaining comparable performance when tested over novel trajectories that are not part of the training environment.

Analysis of the approach

Overall it seems that the *Full NN gain tuner* has very similar performance when compared to the *Model gain tuner* method, as both are able to be adaptive to sensor accuracy and grip conditions, allowing them to outperform the other reinforcement learning approaches. This is not surprising considering the close results shown previously, and considering that the speed control is able to adapt to additional information not available to the *Model gain tuner* method in the previous chapter.

It seems as such that a higher burden has been placed on the speed variation in the *Model gain tuner* method, as only the speed can be used to adapt the behavior of the robot, as opposed to

the *Full NN gain tuner* method. Furthermore, as the *Model gain tuner* is based on the same dynamic model used in the simulation, a high performance is not only expected but suspicious as it could mean that the *Model gain tuner* is exploiting a systemic difference not present in real world conditions. A systemic difference that might have been ignored by the *Full NN gain tuner* method during training.

As such, a real world comparison is needed in order to fully compare both method, and conclude on the effectiveness of these method for adapting the mobile robot's behavior in complex environments.

The results shown in the following section explores the steering control methods previously described, on the RobuFAST platform, in real world conditions.

6.4 Real world experiments

Experimental setup

In these experiments, the goal is to validate the simulated results are translatable to real world experiments, and that the dynamic effect present in the real world experiment is correctly described in the simulation. For this, the RobuFAST robotic platform detailed in section 5.3 is used for the experiments.

The Trajectories

The platform was tested on two novel trajectories (Fig 6.16), that were not available during the training process. This is done to illustrate the generalizing capabilities of the methods to unseen trajectories. These trajectories were defined on location at the Montoldre experimental site,

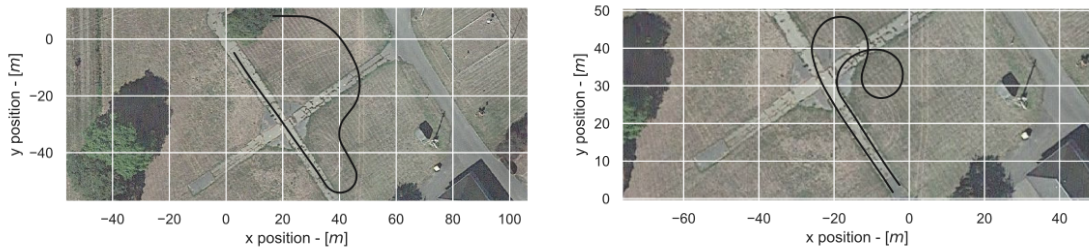


Figure 6.16: Both trajectories on a x,y global reference. On the left, the first trajectory. On the right, the second trajectory.

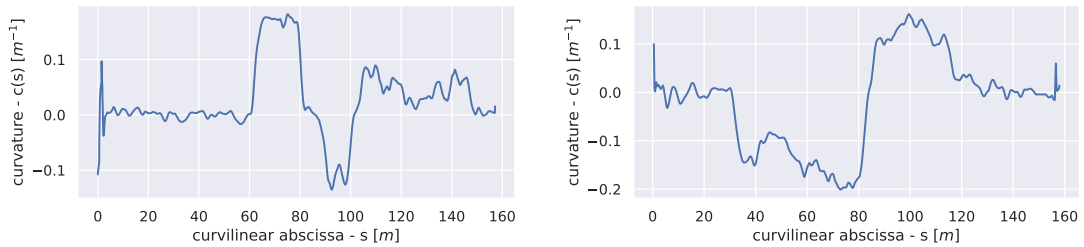


Figure 6.17: The curvature of both trajectories. On the left, the first trajectory. On the right, the second trajectory.

which contains large terrains with variable ground types. Both trajectories contain a mixture of grass and concrete for diversified ground types, and both trajectories are susceptible to naturally occurring GPS losses, allowing for a sensor accuracy drop to occur. These experiments were done over the span of 3 days.

Metrics

The objective function used is the one previously defined for this chapter, shown in the equation (6.5).

$$obj_3 = \max(obj_{err} + k_{steer}obj_{steer}, k_v obj_{speed}) + \gamma(obj_{err} + k_{steer}obj_{steer} + k_v obj_{speed})$$

Since the task remains identical from the previous sections, the metric for the surface error outside the valid corridor is used from the equation (6.6):

$$A_{over} = \sum_{i=0}^N \left| v_i \cos(\tilde{\theta}_i) \left(y_i + \frac{v_i \sin(\tilde{\theta}_i) \Delta t}{2} \right) \mathbb{1}_{|y_i| > y_{lim}} \right| \Delta t$$

Furthermore, the speed metric is also preserved from the equation (6.7):

$$\bar{v} = \frac{1}{T_N} \sum_{i=0}^N |v_i| \Delta t$$

Real world results

Trajectory 1

When focusing with a qualitative analysis over the trajectory 1, with an initial error and a maximal speed of $8m.s^{-1}$, the following results are obtained.

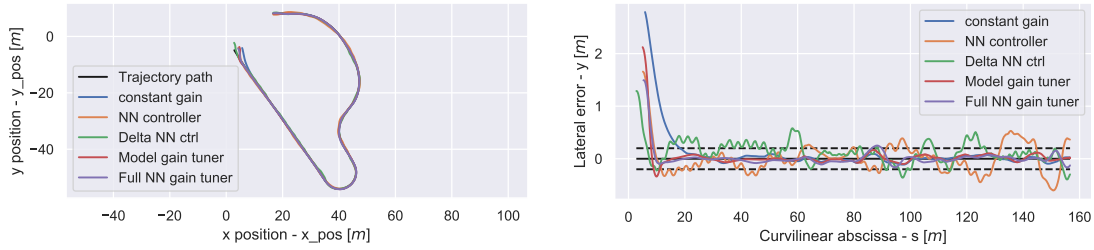


Figure 6.18: On the left: The *first* trajectory on a x, y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.

Figure 6.18 shows that as opposed to the previous section, only the *Full NN gain tuner* and *Model gain tuner* methods are capable of staying within the error corridor. This implies that the *NN controller* and *Delta NN ctrl* methods struggled to transfer the learned behavior across from the simulation to real world conditions due to the systemic error.

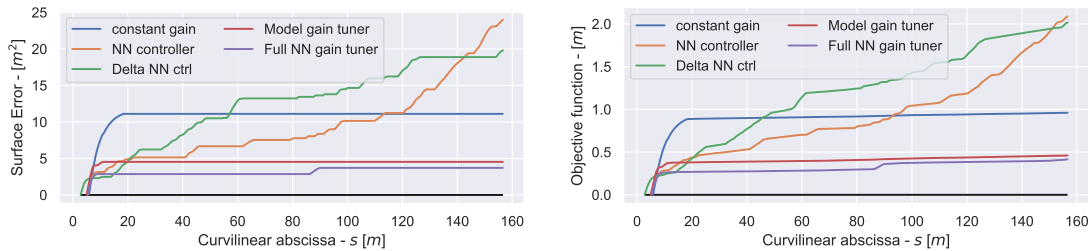


Figure 6.19: On the left: The surface corridor error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.

Figure 6.19 confirms this, and shows that the *Full NN gain tuner* and *Model gain tuner* remain comparable, and have a lower error than the constant gain & speed method.

When observing the speeds on Figure 6.20, the average speeds are very comparable between the *Full NN gain tuner*, *Model gain tuner*, *NN controller*, and the *constant* method, where as the *Delta NN ctrl* method obtained the lowest speed of all. Furthermore, the position accuracy shows that no significantly long signal loss occurred for the GPS sensor.

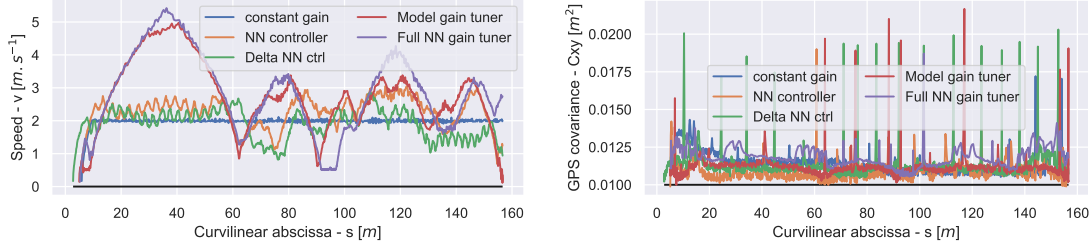


Figure 6.20: On the left: The speed over the curvilinear abscissa. On the right: The position accuracy over the curvilinear abscissa.

Trajectory 2

When focusing with a qualitative analysis over the trajectory 2, with an initial error and a maximal speed of $8m.s^{-1}$, the following results are obtained.

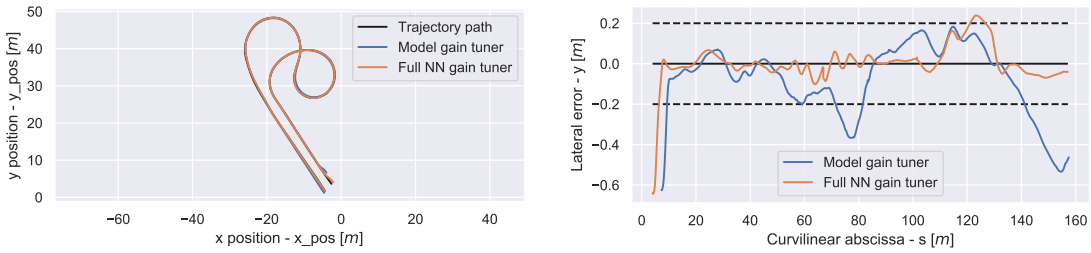


Figure 6.21: On the left: The *second* trajectory on a x,y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.

Figure 6.21 shows that only the *Full NN gain tuner* method is capable of staying within the error corridor, with the *Model gain tuner* reaching error beyond $0.4m$, twice the maximum error corridor. This is likely due to the high curvatures, with strong transitions that lead the *Model gain tuner* with the speed tuning to exceed the expected errors. This can be explained as the longitudinal sliding effect and the weight displacement effect are not modeled in the simulation, which would explain this higher error in real world conditions. This implies that the *NN controller* and *Delta NN ctrl* methods were not able to transfer the learned behavior across from the simulation to real world conditions due to the systemic error, as such they are not shown for the second trajectory.

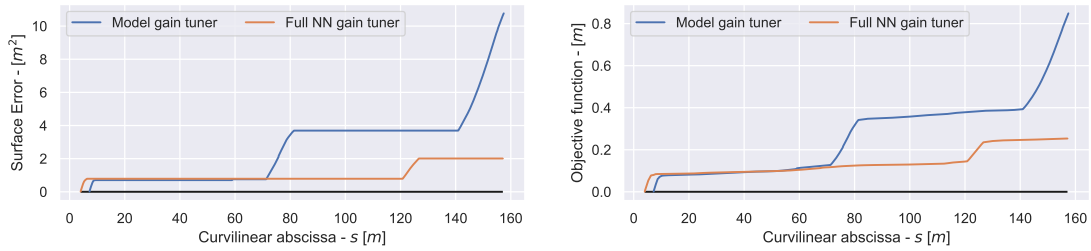


Figure 6.22: On the left: The surface corridor error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.

A result that is clearly confirmed in the Figure 6.22, as the corridor surface error of the *Full NN gain tuner* is around half of the corridor surface error of the *Model gain tuner*, even when the last straight line is considered.

When considering the speed, the *Full NN gain tuner* has a much lower speed when compared to the *Model gain tuner* in the transitions of the corners, which means that the errors are naturally lower, preventing an out of corridor error. Furthermore, the position accuracy plotted

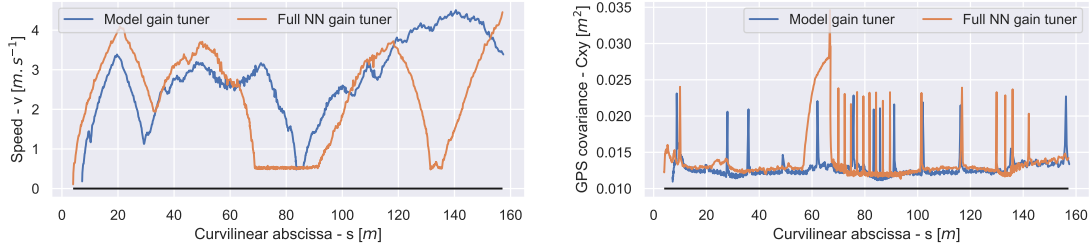


Figure 6.23: On the left: The speed over the curvilinear abscissa. On the right: The position accuracy over the curvilinear abscissa.

shows that a GPS loss occurred during the strongest transition over the curvature for the *Full NN gain tuner* method, which did not cause the method to leave the allowed error corridor.

Analysis of the results

Overall, when transitioning from a simulated environment to a real world environment, a few key aspects could be observed:

- First, each method suffered due to the systemic difference, either on the speed being lower, or the error being higher or both. Nevertheless the methods using parameter tuning are able to achieve behavior close to what is expected.
- Secondly, in real world conditions it seems that the order of the methods from worst to best is *Delta NN ctrl*, *NN controller*, *constant gain*, *Model gain tuner*, and *Full NN gain tuner*, which is not expected as the *Delta NN ctrl* method outmatched the *NN controller* and *constant gain* method in the simulated environment.
- Thirdly, the *Full NN gain tuner* obtained results that are very close to the *Model gain tuner*. Which shows that if an existing system is present, then using RL might not improve it and it should be verified thoroughly. This further implies that replacing existing control systems might not be optimal in all cases.
- Finally, it seems that the "hands off" approach of assisting existing controllers without directly altering its output yielded the best performance in general, and still remained very transferable between simulation and real world conditions.

These aspects are significant, as it means that as expected simulated results by them selves are not sufficient for predicting real world performance in practice. Furthermore, it demonstrates the importance of the adaptability of the control methods, as the real world system was dependent on the weather, terrain type, and GPS quality which all hamper significantly the control methods that are not adapting they behavior to them.

These observations are supported by supplementary experimental results, provided in the appendix section A.9. This section only shown the most representative results of the numerous field experiments achieved during the PhD. Numerous experimental trials have been achieved during the PhD. The results provided in this section only shows the most representative trials, allowing an objective quantitative analysis².

²As a few trials needed to be discarded due to implementation bugs.

Chapter 7

Conclusion and Future works

7.1 Conclusions

The initial question posed by this thesis was "Are machine learning methods able to outperform or improve existing control laws for wheeled mobile robots in complex environments?". The aim is to establish whether a hybrid AI control approach is capable of adapting the behavior of an existing controller, using an AI to enhance said controller.

First, a machine learning approach needed to be determined. Indeed, many methods in machine learning could be considered. When describing the methods in the state of the art of reinforcement learning, a few obstacles occurred as the time difference reinforcement learning methods were not able to converge over the desired path tracking task with the experimental platforms. This was due to a triad of issues. The first was an observability issue as it is not trivial to accurately predict the grip conditions for this system in real time. The second was the action delay from actuators which caused a credit attribution problem. And the third was the inertia of the robot causing all control inputs to be low pass filtered which hampered the exploration of these methods. Some of these elements could be corrected, allowing these methods to train and converge over the task, which can be considered for a future task.

Furthermore, the gradient methods struggled to converge as they have to determine the gradient by considering the observers, state estimators and the robot model in order to calculate the gradient, which means that the resulting gradient used by the neural network is considerably noisy. Which implies that the remaining methods that can work are the direct policy search methods that are gradient free.

Out of those methods explored, CMA-ES returned the best performance in simulated testing. However, as gradient free optimizers are used, careful consideration needed to be taken for configuring the neural network, so that it would not suffer from the curse of dimensionality. For CMA-ES, it was determined that three hidden layers of 64, 128, and 32 perceptrons was ideal, as this achieved a good balance between a low number of parameters (approx 15000), and good performance in our use case, and as such is the neural network used in the rest of the work. However, other architectures that are not fully connected can be used with this configuration, but such methods can be considered for future tasks.

With the appropriate machine learning method chosen, 3 strategies were explored for implementing the AI: The first was the baseline method where the control law was completely replaced with a neural network; the second was a corrective output from the neural network to the control law's steering output; and the third was a gain tuning approach.

From training the methods, it became clear that the existing control laws could be improved using the same inputs directly available to them. Furthermore, from the explored methods of integrating the neural network's output to the control loop, it was determined that controlling the steering output directly yielded sub-optimal results, as it was outperformed by the method correcting the steering output. The corrective steering output method was itself then outperformed by the gain and horizon tuning method. This showed that letting the neural network only control the gains and horizon (which in turn reduced the impact of the neural network), increased the performance by making it easy to adjust each gain with respect to the different inputs. (See

Figure 4.18).

The method that directly controls the steering output gave some interesting insights in simulation, as the inputs used by the method seem to match the design of the existing control laws, and it adapted to parameters that the theory suggests are useful (See Figure 4.5). Furthermore, controlling the steering output directly caused a strong training difficulty, which can be a problem if the task is too hard, and so an alternative to CMA-ES might need to be explored if this occurs.

Furthermore, it was shown that the adaptive gain system based on a neural network allowed for better performance not only when changes in the environment occur, but also in a nominal case as it is able to fully exploit the loop information when it does not need to be conservative over the reactivity. This showed that the gain tuning approach allows for enough modulation from the NN, without overriding the control law completely, which led to a higher performance when tested.

It was noted that comparing a gain tuning NN to other NN approaches was not sufficient, as gain tuning methods do exist for control laws, even though it is a challenging approach. Therefore, a model-based gain tuning method was developed for comparison, with dynamic parameters as an additional input. From this, it was shown that the control parameter tuning is able to adapt the behavior of the robot to the environment (as shown with the model gain tuning method), using only the gains and not affecting directly the steering. Furthermore, the neural network based gain and horizon tuning method was able to match and even exceed the model based gain tuning in some cases.

However, it was shown that the neural network gain and horizon tuning method is sensitive to the input, as the addition of inputs that lead to an incomplete internal model can degrade performance. Furthermore, the neural network seemed limited by the information it could be given. Indeed, if more information is available it would improve its performance, such as using data from a camera, or improving the cornering stiffness observer. It was shown that pre-calculating the cornering stiffness is useful for the neural network, even if the same inputs are available and the neural network could rebuild this information if needed. This means that there is a clear advantage to spending engineering time in order to correctly feed the neural network with relevant computed inputs.

During experiments, it was noted that a maximum speed of $4m.s^{-1}$ was the upper limit for the training, due to the grip conditions and trajectories being nearly non admissible dynamically over $5m.s^{-1}$. As such, experiments with the tuning of the speed along with the hybrid steering approach were conducted. Adding the speed to the steer control is not a trivial task, as it requires integration as a function of curvilinear abscissa, and not a simple time integration. Furthermore, the method is dependent on a good design of the objective function (see section 6.1 about Pareto fronts).

A first approach had consisted in using neural networks only for speed control while cooperating with a deterministic control, and a model based gain tuning adaptation. The thesis showed that the neural network tuning only the speed can help to adapt the steering to errors, GPS losses, and grip conditions (see Figure 6.14). This shows that an existing method of gain tuning is as good as a neural network based method under ideal circumstances, showing that the neural networks are not the only valid approach for solving these types of problems.

Important conclusions from the real-world experiments are that more complex adaptive neural network systems are sensitive to systemic error, and that the greater the control and dependence of the neural network on the control output, the more likely it is that it will over-adapt to the simulated environment, leading to significant problems in real-world conditions. This indicates that the parameter tuning neural network has good transferability to real-world conditions, when compared to direct steering control using a neural network.

The field experiments also revealed significant problems, as the methods are limited to the simulated model, which does not take into account the weight displacement and longitudinal sliding of the physical system. A neural network hybrid model or a more complex model could be envisioned for simulation training.

Overall, when answering the initial question: Adapting the control parameters is not a trivial task, as they often depend on variables that can only be known in real time. For this, control laws have been often tuned to match the expected value of these variables, leading to sub-optimal

performance. The work presented previously shows the capacity of machine learning methods, in particular neural networks tuned by evolutionary strategies, to determine control parameters in real time, that out perform the model based control parameters methods and constant control parameters methods. These methods have shown to be able to find an effective mapping between the observed robotic state and the quasi-optimal control parameters.

It has been shown to be able to adapt to changes in the environment, encoded through the state estimator and observers, such that the method improves the task performance in real world experimentation without destabilizing the robot. We have also validated the importance of the observed values (sliding angles and cornering stiffness) with respect to the performance, and showed that including this information allows for higher performance when compared without.

These methods have shown limitations with respect to the simulated model. As it has been shown previously, higher speed means more dynamic phenomena which is harder to accurately model. For this, more work is required to close the reality gap, such as a more complex simulated model.

All these comments are derived from simulated and experimental results achieved in different conditions (soil, trajectory, weather, ...). During this thesis, 20 days of real world experimental trials have been achieved, in order to validate and affirm the results presented in this PhD.

Furthermore, these methods have the capacity to generalize to different types of control parameters such as the horizon of a predictive controller, or to generalize to different types of gains such as the observation gains. All these aspects show the promising capabilities of the methods, hence the future works that can be envisioned and that are currently being considered.

7.2 Future works & perspectives

From these conclusions, a few paths can be observed on the horizon for future works.

Tuning the model based gain tuner

When constructing the model based gain tuner, two strong hypothesis were implied. The first was that the optimal dampening of the system ξ is always 1. And the second was that the sampling frequency N satisfying the Shannon condition was constant. However, as shown by the neural network gain tuning methods this is not the case. As such, the value of N which is proportional to the K_p gain, and the dampening ξ which is proportional to the K_d gain, can be modulated in real time by a neural network, allowing for a higher level of tuning of the control system. This is very important, as it is the process of trying to find novel ways of integrating neural networks into existing and complex control system in a symbiotic way, that further insights into neural networks and control laws can be determined, through observing which inputs are used and how the neural network learns. With this, the stepping stones needed in order to control complex robots for complex tasks can be envisioned to be realistically attainable.

Improving the observations

One of the limiting factors of the approach used up until now, has been the observability of certain dynamic elements of the system. As such, the addition of a sensor or a more complex observer can be considered, in order to provide more accurate information to the neural network, which in turn allows for a more accurate path tracking. Indeed, additional information that is critical to predict the future behavior of the system (such as using a camera to observe the terrain quality in front of the robot) is required, however this information can only be acquired via additional sensors. The interpretation of these sensors or their preinterpreted information could be used to allow for higher performance.

Alternate architecture for integrating the neural network

The neural network has been integrated a total of three different ways in this work. However, it is clearly not limited to this, and additional ways of integrating neural network can and should be explored. For example a neural network that acts as a Selector network for two existing controllers, allowing for specialized control laws to be used in generalized control tasks.

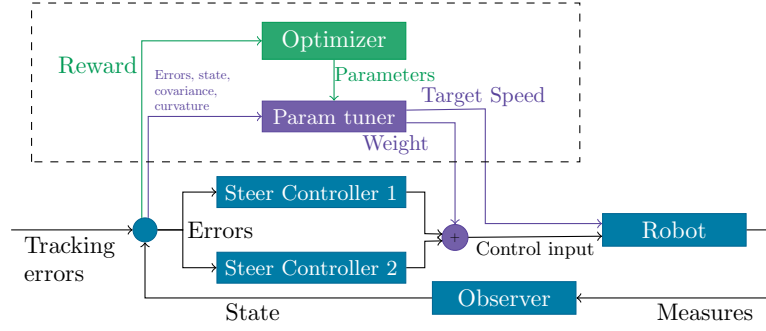


Figure 7.1: Overview of the proposed method.

In this case, special consideration should be taken for the objective function, in order to guarantee that the neural network will achieve smooth transitions between the control laws (or an additional output from the neural network could be added to address this), where stability and smooth control are desired above performance.

Improving the simulation for additional dynamics

As seen in the results, if the speed is varied over time, then the model used is no longer valid as it does not take into account any longitudinal dynamics. Furthermore, it does not take into account any weight displacement due to acceleration and deceleration, which can vary the steering characteristics of the robot. As such, future works are planned with a more realistic model. A possible alternative to consider, could be to complete the dynamic model, using a supervised neural network. Where data from real world conditions could be measured, and used to predict the future state in a supervised manor directly. Care needs to be taken in order to avoid the noise permeating the data, such as using a 7 point derivative.

Predicting the settling time with a neural network, for agnostic controller gain tuning

Given a damping factor ξ , and with a settling distance D_y for the convergence of the lateral error, a valid approximation of the control gains can be inferred for some controllers. As such, future works include training a neural network to predict these values for a given controller, and then test a different controller without retraining the neural network. Indeed we have seen that gains are setting the distance of convergence for the control laws. As a result, instead of tuning parameters of a deterministic control law, we can tune more general parameters that can infer the gains, which allows for a level of abstraction with respect to the control law, and could permit the tuning of multiple control laws from a single neural network.

Gain tuning: going further than controllers

One of the aspects that is being worked on is to alter the controller gain prediction system developed during this PhD, in order to apply it to observer. Indeed, observer are also dependent on their parameters which are often set a priori and are constant. However, these parameters affect the behavior of the observer, and in some situations changing these values are needed (i.e. changing reactivity of the observer when the noise increases). Special care will need to be taken, as the validation of the observers is considerably harder than validating control laws. Furthermore, it is not clear if all observer are tunable in real time, without adverse effects. An alternative could be to use a machine learning method to predict the observed parameters (such as β_F or C_R), in order to correct or replace existing observers.

Speed control applied independently to each wheel

For some mobile robots, each wheel is independently controllable by a motor, even if usually controlled in a symmetric manner for simplicity in control. Instead, this could be achieved using the neural network based speed control developed here with slight modifications. Indeed, instead

of tuning a single speed, the neural network could output an independent speed for each wheel. This could in turn allow for complex dynamic behavior, including slippage correction for higher computed speeds, or even drift control.

Transformer applied to robotic control

The transformer architecture [99] is a method of using attention neural networks, in order to predict an output from a sequence of elements. In a robotic context, this could be used in order to pull specific data point from previous timesteps, and use them in order to determine the optimal output. Furthermore, the attention mechanism is not opaque, and as such can be used as an additional tool for interpreting the neural networks. However, this would increase the complexity of the neural network, which would make its predictability and interpretability far more complicated. Furthermore, the number of parameters of the neural network would increase considerably, which means an alternative to CMA-ES would need to be considered.

Custom Neural network architecture

Rather than only tune the weights from a fully connect network, one could consider the tuning of the neural network architecture as well, in order to determine an optimal structure a priori for the task. This would mimic nature, and allow for the neural network to have an implicitly behavior that is "burnt-in" by the underlying structure of the neural network. Indeed, one could consider that the steering angle be dependent on the opposite value of the angular error, and directly "hard-wire" the neurons together in order to encourage this behavior. However, doing this over the whole network would be complex. Which is why an optimizer could be considered to achieve this, such as what is done in [100, 101].

Improving the optimizer

One of the issues raised in the conclusion is the CMA-ES optimizer. It is not sample efficient, and it is sensitive to large neural network due to the curse of dimensionality when randomly sampling in a large search space. As such, two choices can be considered here: Either augment CMA-ES in order to reduce these issues, and in turn improve the performance of the obtained results and being able to increase the size of the neural network. Or fix the triad issues of the time difference reinforcement learning, as detailed in section 3. Fundamentally, solving this issue would allow for an increase in the neural network size, to improve the training time, and to improve the performance of the final neural network.

7.3 Overview of the work

We showed that episodic reinforcement learning methods for policy iteration with neural networks can be used in order to increase the adaptability of autonomous mobile robots. We demonstrated that using such techniques to adapt model based control improved their performances, and outperformed end-to-end approaches. Furthermore, these techniques allowed for adapting the robot to complex inputs, such as the terrain and the sensor accuracy, in real time.

This opens the way to developing new strategies for increasing the adaptability and versatility of autonomous robots, by a judicious cooperation between model based controllers and machine learning based AI systems, rather than using them as opposing approaches.

Bibliography

- [1] A. M. Turing, “Computing machinery and intelligence,” in *Parsing the turing test*. Springer, 2009, pp. 23–65.
- [2] E. B. Braaten and D. Norman, “Intelligence (iq) testing,” *Pediatrics in review*, vol. 27, no. 11, pp. 403–408, 2006.
- [3] W. Weiten, *Psychology: Themes and variations*. Cengage Learning, 2021.
- [4] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, “Mastering the game of go without human knowledge,” *nature*, vol. 550, no. 7676, pp. 354–359, 2017.
- [5] E. Hubinger, C. van Merwijk, V. Mikulik, J. Skalse, and S. Garrabrant, “Risks from learned optimization in advanced machine learning systems,” 2019. [Online]. Available: <https://arxiv.org/abs/1906.01820>
- [6] J. Koch, L. Langosco, J. Pfau, J. Le, and L. Sharkey, “Objective robustness in deep reinforcement learning,” 2021. [Online]. Available: <https://arxiv.org/abs/2105.14111>
- [7] S. J. Russell, *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.
- [8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.
- [9] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [10] K. Hornik, M. Stinchcombe, and H. White, “Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks,” *Neural Networks*, vol. 3, no. 5, pp. 551 – 560, 1990. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/0893608090900056>
- [11] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.
- [12] C. Ramos, J. C. Augusto, and D. Shapiro, “Ambient intelligence—the next step for artificial intelligence,” *IEEE Intelligent Systems*, vol. 23, no. 2, pp. 15–18, 2008.
- [13] J. J. Hopfield and D. W. Tank, ““neural” computation of decisions in optimization problems,” *Biological cybernetics*, vol. 52, no. 3, pp. 141–152, 1985.
- [14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, vol. 323, no. 6088, pp. 533–536, 1986.
- [15] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, “Flexible, high performance convolutional neural networks for image classification,” in *Twenty-second international joint conference on artificial intelligence*, 2011.
- [16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
- [17] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
- [18] G. Tesauro *et al.*, “Temporal difference learning and td-gammon,” *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.
- [19] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, “Playing atari with deep reinforcement learning,” *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: <http://arxiv.org/abs/1312.5602>
- [20] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas *et al.*, “Solving rubik’s cube with a robot hand,” *arXiv preprint arXiv:1910.07113*, 2019.

- [21] D. A. Pomerleau, “Alvin: An autonomous land vehicle in a neural network,” *Advances in neural information processing systems*, vol. 1, 1988.
- [22] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann *et al.*, “Stanley: The robot that won the darpa grand challenge,” *Journal of field Robotics*, vol. 23, no. 9, pp. 661–692, 2006.
- [23] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer *et al.*, “Autonomous driving in urban environments: Boss and the urban challenge,” *Journal of field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.
- [24] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, “End to end learning for self-driving cars,” 2016. [Online]. Available: <https://arxiv.org/abs/1604.07316>
- [25] X. Pan, Y. You, Z. Wang, and C. Lu, “Virtual to real reinforcement learning for autonomous driving,” 2017. [Online]. Available: <https://arxiv.org/abs/1704.03952>
- [26] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: <http://arxiv.org/abs/1602.01783>
- [27] Q. Khan, T. Schön, and P. Wenzel, “Latent space reinforcement learning for steering angle prediction,” 2019. [Online]. Available: <https://arxiv.org/abs/1902.03765>
- [28] J. C. Gerdes, “Neural networks overtake humans in gran turismo racing game,” 2022.
- [29] M. Yan, I. Frosio, S. Tyree, and J. Kautz, “Sim-to-real transfer of accurate grasping with eye-in-hand observations and continuous control,” 2017. [Online]. Available: <https://arxiv.org/abs/1712.03303>
- [30] S. R. Richter, H. A. AlHaija, and V. Koltun, “Enhancing photorealism enhancement,” *arXiv:2105.04619*, 2021.
- [31] D. Giglio, “Task scheduling for multiple forklift agvs in distribution warehouses,” in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. IEEE, 2014, pp. 1–6.
- [32] L. Li, Y.-H. Liu, M. Fang, Z. Zheng, and H. Tang, “Vision-based intelligent forklift automatic guided vehicle (agv),” in *2015 IEEE International Conference on Automation Science and Engineering (CASE)*. IEEE, 2015, pp. 264–265.
- [33] A. Michaels, S. Haug, and A. Albert, “Vision-based high-speed manipulation for robotic ultra-precise weed control,” in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2015, pp. 5498–5505.
- [34] J. Rohde, J. E. Stellet, H. Mielenz, and J. M. Zöllner, “Localization accuracy estimation with application to perception design,” in *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2016, pp. 4777–4783.
- [35] T. Peynot, J. Underwood, and S. Scheduling, “Towards reliable perception for unmanned ground vehicles in challenging conditions,” in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2009, pp. 1170–1176.
- [36] G. B. Margolis, G. Yang, K. Paigwar, T. Chen, and P. Agrawal, “Rapid locomotion via reinforcement learning,” 2022. [Online]. Available: <https://arxiv.org/abs/2205.02824>
- [37] F. B. Amar, C. Grand, G. Besseron, D. Lhomme-Desages, E. Lucet, and P. Bidaud, “Mobility and stability of robots on rough terrain: modeling and control,” in *Proceedings of IROS’08 Workshop on Modeling, Estimation, Path Planning and Control of All Terrain Mobile Robots*, 2008, pp. 5–11.
- [38] D. Filliat, “Robotique mobile,” Ph.D. dissertation, EDX, 2011.
- [39] G. Campion, G. Bastin, and B. Dandrea-Novel, “Structural properties and classification of kinematic and dynamic models of wheeled mobile robots,” *IEEE transactions on robotics and automation*, vol. 12, no. 1, pp. 47–62, 1996.
- [40] B. Siciliano and O. Khatib, *Springer handbook of robotics*. Springer, 2016.
- [41] D. Schramm, M. Hiller, and R. Bardini, “Vehicle dynamics,” *Modeling and Simulation. Berlin, Heidelberg*, vol. 151, 2014.
- [42] P. Polack, F. Althé, B. d’Andréa Novel, and A. de La Fortelle, “The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles?” in *2017 IEEE intelligent vehicles symposium (IV)*. IEEE, 2017, pp. 812–818.

- [43] L. Li, “Modélisation et contrôle d’un véhicule tout-terrain à deux trains directeurs,” Ph.D. dissertation, Université Paris sciences et lettres, 2021, thèse de doctorat dirigée par D’andrea-Novel, Brigitte Informatique temps réel, robotique et automatique Université Paris sciences et lettres 2021. [Online]. Available: <http://www.theses.fr/2021UPSLM032>
- [44] F. B. Amar, P. Jarrault, P. Bidaud, and C. Grand, “Analysis and optimization of obstacle clearance of articulated rovers,” in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2009, pp. 4128–4133.
- [45] G. Rill, *Road vehicle dynamics: fundamentals and modeling*. Crc Press, 2011.
- [46] C. C. De Wit and P. Tsiotras, “Dynamic tire friction models for vehicle traction control,” in *Proceedings of the 38th IEEE conference on decision and control (Cat. no. 99CH36304)*, vol. 4. IEEE, 1999, pp. 3746–3751.
- [47] D. Lhomme-Desages, “Commande d’un robot mobile rapide à roues non directionnelles sur sol naturel,” Ph.D. dissertation, Université Pierre et Marie Curie-Paris VI, 2008.
- [48] E. Bakker, L. Nyborg, and H. B. Pacejka, “Tyre modelling for use in vehicle dynamics studies,” in *SAE Technical Paper*. JSTOR, 1987, pp. 190–204.
- [49] M. Deremetz, “Contribution à la modélisation et à la commande de robots mobiles autonomes et adaptables en milieux naturels,” Ph.D. dissertation, Université Clermont Auvergne, 2018, thèse de doctorat dirigée par Lenain, Roland Robotique Université Clermont Auvergne (2017-2020) 2018. [Online]. Available: <http://www.theses.fr/2018CLFAC079>
- [50] R. Lenain, B. Thuilot, C. Cariou, and P. Martinet, “Adaptive and predictive path tracking control for off-road mobile robots,” *European journal of control*, vol. 13, no. 4, pp. 419–439, 2007.
- [51] R. Lenain, M. Deremetz, J.-B. Braconnier, B. Thuilot, and V. Rousseau, “Robust sideslip angles observer for accurate off-road path tracking control,” *Advanced Robotics*, vol. 31, no. 9, pp. 453–467, 2017.
- [52] E. Lucet, A. Micaelli, and F.-X. Russotto, “Accurate autonomous navigation strategy dedicated to the storage of buses in a bus center,” *Robotics and Autonomous Systems*, vol. 136, p. 103706, 2021.
- [53] G. Welch, G. Bishop *et al.*, “An introduction to the kalman filter,” 1995.
- [54] W. H. Press, W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, and W. T. Vetterling, *Numerical recipes in Pascal: the art of scientific computing*. Cambridge university press, 1989, vol. 1.
- [55] R. Alexander, “Solving ordinary differential equations i: Nonstiff problems (e. hairer, sp norsett, and g. wanner),” *Siam Review*, vol. 32, no. 3, p. 485, 1990.
- [56] J. R. Dormand, *Numerical methods for differential equations: a computational approach*. CRC press, 1996, vol. 3.
- [57] D. P. Kingma and M. Welling, “Stochastic gradient vb and the variational auto-encoder,” in *Second International Conference on Learning Representations, ICLR*, vol. 19, 2014.
- [58] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in *Advances in neural information processing systems*, 2014, pp. 2672–2680.
- [59] T. Lesort, N. Díaz-Rodríguez, J.-F. Goudou, and D. Filliat, “State representation learning for control: An overview,” *Neural Networks*, vol. 108, pp. 379–392, 2018.
- [60] A. Raffin, A. Hill, R. Traoré, T. Lesort, N. Díaz-Rodríguez, and D. Filliat, “Decoupling feature extraction from policy learning: assessing benefits of state representation learning in goal based robotics,” *arXiv preprint arXiv:1901.08651*, 2019.
- [61] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: <http://arxiv.org/abs/1707.06347>
- [62] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” *CoRR*, vol. abs/1509.02971, 2015. [Online]. Available: <http://arxiv.org/abs/1509.02971>
- [63] R. S. Sutton, “Learning to predict by the methods of temporal differences,” *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

- [64] P. L. Bartlett and W. Maass, "Vapnik-chervonenkis dimension of neural nets," *The handbook of brain theory and neural networks*, pp. 1188–1192, 2003.
- [65] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [66] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *CoRR*, vol. abs/1801.01290, 2018. [Online]. Available: <http://arxiv.org/abs/1801.01290>
- [67] A. Hill, A. Raffin, M. Ernestus, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines," <https://github.com/hill-a/stable-baselines>, 2018.
- [68] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.
- [69] M. Minsky, "Steps toward artificial intelligence," *Proceedings of the IRE*, vol. 49, no. 1, pp. 8–30, 1961.
- [70] A. Raffin and F. Stulp, "Generalized state-dependent exploration for deep reinforcement learning in robotics," *arXiv preprint arXiv:2005.05719*, 2020.
- [71] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," *ArXiv e-prints*, 2017.
- [72] D. Wierstra, T. Schaul, J. Peters, and J. Schmidhuber, "Natural evolution strategies," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, 2008, pp. 3381–3387.
- [73] A. Auger and N. Hansen, "A restart cma evolution strategy with increasing population size," in *2005 IEEE congress on evolutionary computation*, vol. 2. IEEE, 2005, pp. 1769–1776.
- [74] A. S. Fukunaga, "Restart scheduling for genetic algorithms," in *International Conference on Parallel Problem Solving from Nature*. Springer, 1998, pp. 357–366.
- [75] H. Mania, A. Guy, and B. Recht, "Simple random search provides a competitive approach to reinforcement learning," 2018.
- [76] F. Stulp and O. Sigaud, "Path integral policy improvement with covariance matrix adaptation," *arXiv preprint arXiv:1206.4621*, 2012.
- [77] N. Hansen, A. Auger, R. Ros, S. Finck, and P. Pošík, "Comparing results of 31 algorithms from the black-box optimization benchmarking bbob-2009," in *Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation*, ser. GECCO '10. New York, NY, USA: ACM, 2010, pp. 1689–1696. [Online]. Available: <http://doi.acm.org/10.1145/1830761.1830790>
- [78] N. Hansen, "The CMA evolution strategy: A tutorial," *CoRR*, vol. abs/1604.00772, 2016.
- [79] R. Ros and N. Hansen, "A simple modification in cma-es achieving linear time and space complexity," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2008, pp. 296–305.
- [80] S. K. Kumar, "On weight initialization in deep neural networks," *CoRR*, vol. abs/1704.08863, 2017. [Online]. Available: <http://arxiv.org/abs/1704.08863>
- [81] B. L. Welch, "The generalization of 'student's' problem when several different population variances are involved," *Biometrika*, vol. 34, no. 1-2, pp. 28–35, 01 1947.
- [82] J. G. Ziegler, N. B. Nichols *et al.*, "Optimum settings for automatic controllers," *trans. ASME*, vol. 64, no. 11, 1942.
- [83] F. Loucif, S. Kechida, and A. Sebbagh, "Whale optimizer algorithm to tune pid controller for the trajectory tracking control of robot manipulator," *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, vol. 42, no. 1, pp. 1–11, 2020.
- [84] C.-S. Chiu, K.-Y. Lian, and P. Liu, "Fuzzy gain scheduling for parallel parking a car-like robot," *IEEE Transactions on Control Systems Technology*, vol. 13, no. 6, pp. 1084–1092, 2005.
- [85] K.-L. Han, O.-K. Choi, J. Kim, H. Kim, and J. S. Lee, "Design and control of mobile robot with mecanum wheel," in *2009 ICCAS-SICE*. IEEE, 2009, pp. 2932–2937.
- [86] S. Khesrani, A. Hassam, M. Boubezoula, and F. Srairi, "Modeling and control of mobile platform using flatness-fuzzy based approach with gains adjustment," in *2017 6th International Conference on Systems and Control (ICSC)*. IEEE, 2017, pp. 173–177.

- [87] J.-B. He, Q.-G. Wang, and T.-H. Lee, "Pi/pid controller tuning via lqr approach," *Chemical Engineering Science*, vol. 55, no. 13, pp. 2429–2439, 2000.
- [88] L. M. Argentim, W. C. Rezende, P. E. Santos, and R. A. Aguiar, "Pid, lqr and lqr-pid on a quadcopter platform," in *2013 International Conference on Informatics, Electronics and Vision (ICIEV)*. IEEE, 2013, pp. 1–6.
- [89] M. Deremetz, R. Lenain, B. Thuilot, and V. Rousseau, "Adaptive trajectory control of off-road mobile robots: A multi-model observer approach," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 4407–4413.
- [90] F. Altche, P. Polack, and A. de La Fortelle, "High-speed trajectory planning for autonomous vehicles using a simple dynamic model," in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. Yokohama: IEEE, Oct. 2017, pp. 1–7.
- [91] C. Gámez Serna and Y. Ruichek, "Dynamic Speed Adaptation for Path Tracking Based on Curvature Information and Speed Limits," *Sensors*, vol. 17, no. 6, p. 1383, Jun. 2017, number: 6.
- [92] M. Park, S. Lee, and W. Han, "Development of Steering Control System for Autonomous Vehicle Using Geometry-Based Path Tracking Algorithm," *ETRI Journal*, vol. 37, no. 3, pp. 617–625, Jun. 2015, number: 3.
- [93] J.-B. Braconnier, R. Lenain, and B. Thuilot, "Ensuring path tracking stability of mobile robots in harsh conditions: An adaptive and predictive velocity control," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. Hong Kong, China: IEEE, May 2014, pp. 5268–5273.
- [94] J. Braconnier, R. Lenain, B. Thuilot, and V. Rousseau, "High speed path tracking application in harsh conditions: Predictive speed control to restrict the lateral deviation to some threshold," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 3087–3094.
- [95] O. Hach, R. Lenain, B. Thuilot, and P. Martinet, "Avoiding steering actuator saturation in off-road mobile robot path tracking via predictive velocity control," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. San Francisco, CA: IEEE, Sep. 2011, pp. 4072–4077.
- [96] I. Y. Kim and O. L. De Weck, "Adaptive weighted-sum method for bi-objective optimization: Pareto front generation," *Structural and multidisciplinary optimization*, vol. 29, no. 2, pp. 149–158, 2005.
- [97] C.-L. Hwang and A. S. M. Masud, *Multiple objective decision making—methods and applications: a state-of-the-art survey*. Springer Science & Business Media, 2012, vol. 164.
- [98] D. Golovin and Q. Zhang, "Random hypervolume scalarizations for provable multi-objective black box optimization," *arXiv preprint arXiv:2006.04655*, 2020.
- [99] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.
- [100] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.
- [101] C. D. Freeman, L. Metz, and D. Ha, "Learning to predict without looking ahead: World models without forward prediction," 2019. [Online]. Available: <https://arxiv.org/abs/1910.13038>
- [102] D. Greenhalgh and S. Marshall, "Convergence criteria for genetic algorithms," *SIAM Journal on Computing*, vol. 30, no. 1, pp. 269–282, 2000.
- [103] G. A. Jastrebski and D. V. Arnold, "Improving evolution strategies through active covariance matrix adaptation," in *2006 IEEE international conference on evolutionary computation*. IEEE, 2006, pp. 2814–2821.
- [104] Y. Akimoto, A. Auger, and N. Hansen, "Comparison-based natural gradient optimization in high dimension," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 2014, pp. 373–380.
- [105] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection," *The International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 421–436, 2018.
- [106] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, "Sim-to-real: Learning agile locomotion for quadruped robots," *CoRR*, vol. abs/1804.10332, 2018. [Online]. Available: <http://arxiv.org/abs/1804.10332>

- [107] F. Golemo, A. A. Taiga, A. Courville, and P.-Y. Oudeyer, “Sim-to-real transfer with neural-augmented robot simulation,” in *Proceedings of The 2nd Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, A. Billard, A. Dragan, J. Peters, and J. Morimoto, Eds., vol. 87. PMLR, 29–31 Oct 2018, pp. 817–828. [Online]. Available: <http://proceedings.mlr.press/v87/golemo18a.html>
- [108] A. Liaw, M. Wiener *et al.*, “Classification and regression by randomforest,” *R news*, vol. 2, no. 3, pp. 18–22, 2002.
- [109] S. Suthaharan, “Decision tree learning,” in *Machine Learning Models and Algorithms for Big Data Classification*. Springer, 2016, pp. 237–269.
- [110] C. Molnar, *Interpretable machine learning*. Lulu. com, 2019.
- [111] A. Hill, E. Lucet, and R. Lenain, “A new neural network feature importance method: Application to mobile robots controllers gain tuning,” in *Proceedings of the 17th International Conference on Informatics in Control, Automation and Robotics, ICINCO 2020*. ScitePress, 2020, pp. 188–194. [Online]. Available: <https://doi.org/10.5220/0009888501880194>
- [112] A. Mordvintsev, C. Olah, and M. Tyka, “Deepdream-a code example for visualizing neural networks,” *Google Research*, vol. 2, no. 5, 2015.
- [113] M. T. Ribeiro, S. Singh, and C. Guestrin, “”why should i trust you?”: Explaining the predictions of any classifier,” 2016. [Online]. Available: <https://arxiv.org/abs/1602.04938>
- [114] S. Fujimoto, H. van Hoof, and D. Meger, “Addressing function approximation error in actor-critic methods,” 2018.
- [115] F. Gauthier-Clerc, A. Hill, J. Laneurit, R. Lenain, and E. Lucet, “Online velocity fluctuation of off-road wheeled mobile robots: A reinforcement learning approach,” in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 2421–2427.

List of Figures

1.1	XKCD's spin on Turing tests.	15
1.2	A diagram of a neuron.	16
1.3	Marvin the paranoid robot ("The Hitchhiker's guide to the galaxy")	16
1.4	From <i>Artificial Intelligence: A Modern Approach</i> [7].	17
1.5	The car ALVINN used.	18
2.1	The cinematic robot model.	24
2.2	The dynamic robot model.	25
2.3	The Pacejka model's lateral force curve, using a $F_z = 1.055\text{kN}$, with respect to the tyre slip angle.	26
2.4	An example of the model's delayed steering	27
2.5	The extended cinematic robot model.	28
2.6	An example of the model's delayed steering	31
3.1	An example of a Markov decision process for a "racing car". Each state represents an environmental state and configuration (position, speed, ...), the action influences the next state, and the reward is the quality of the transition between two states.	38
3.2	A block diagram of a control loop using reinforcement learning.	38
3.3	An example of a policy model.	40
3.4	A hierarchical diagram of some of the types of reinforcement learning methods.	42
3.5	The full training loop with the objective function and optimizer.	44
3.6	The CMA Evolution strategy, each point is a sampled population, the cross is the mean, the full circle is the covariance, and the dotted circle is the old covariance. Left: the initial population sampling from the mean and the covariance. Middle: the elitist selection after the evaluation, updated mean and covariance in the direction of the local minimum. Right: New covariance and mean for sampling the next generation	45
3.7	A representation of the trajectories used in training.	46
3.8	The neural network architecture used in the following works.	47
4.1	Overview of the proposed method.	49
4.2	On the left: The <i>spline5</i> trajectory on a x,y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.	53
4.3	On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.	54
4.4	On the left: The real steering state over the curvilinear abscissa. On the right: The steering input from the controller over the curvilinear abscissa.	54
4.5	The feature importance for the <i>NN controller</i> method for each input, denoted in % of importance.	54
4.6	Overview of the proposed method.	56
4.7	On the left: The <i>spline5</i> trajectory on a x,y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.	58
4.8	On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.	59
4.9	On the left: The real steering state over the curvilinear abscissa. On the right: The steering input from the controller over the curvilinear abscissa.	59
4.10	The feature importance for the <i>Delta NN ctrl</i> method for each input, denoted in % of importance.	59

4.11	Example of sources of influence on the optimal control parameters. Left: wheel, actuator dynamics. Middle: GPS sensor, perception quality. Right: ground, environment. . . .	60
4.12	Explainability and adaptability compromise in control parameter tuning for controllers.	62
4.13	Overview of the proposed method.	63
4.14	On the left: The <i>spline5</i> trajectory on a x,y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.	66
4.15	On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.	66
4.16	On the left: The real steering state over the curvilinear abscissa. On the right: The steering input from the controller over the curvilinear abscissa.	66
4.17	On the left: The gains over the curvilinear abscissa. On the right: The horizon over the curvilinear abscissa.	67
4.18	The feature importance for the <i>NN gain tuner</i> method for each input, denoted in % of importance.	67
5.1	Overview of the proposed method.	73
5.2	On the left: The <i>spline5</i> trajectory on a x,y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.	75
5.3	On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.	75
5.4	On the left: The real steering state over the curvilinear abscissa. On the right: The steering input from the controller over the curvilinear abscissa.	76
5.5	On the left: The gains over the curvilinear abscissa. On the right: The horizon over the curvilinear abscissa.	76
5.6	Overview of the proposed method.	77
5.7	On the left: The <i>spline5</i> trajectory on a x,y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.	80
5.8	On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.	80
5.9	On the left: The gains over the curvilinear abscissa. On the right: The control damping over the curvilinear abscissa.	81
5.10	The horizon over the curvilinear abscissa.	81
5.11	On the left: The <i>spline5</i> trajectory on a x,y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.	82
5.12	On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.	82
5.13	On the left: The gains over the curvilinear abscissa. On the right: The control damping over the curvilinear abscissa.	82
5.14	On the left: The <i>spline5</i> trajectory on a x,y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.	83
5.15	On the left: The surface error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.	83
5.16	On the left: The gains over the curvilinear abscissa. On the right: The control damping over the curvilinear abscissa.	83
5.17	The feature importance for the <i>Full NN gain tuner</i> method for each input, denoted in % of importance.	84
5.18	The RobuFAST robotic platform	86
5.19	The trajectories. Left: trajectory 1. Right: trajectory 2.	87
5.20	The trajectory (on the left) and the lateral error (on the right). Over the total trajectory	87
5.21	The surface error A_{error} , and the surface error A_{error} after the initial lateral error. . . .	87
5.22	The objective function, and the objective function after the initial lateral error. . . .	87
5.23	The gains, and the horizon.	88
5.24	The trajectory (on the left), and the lateral error (on the right). Over the total trajectory	88
5.25	The surface error A_{error} , and the surface error A_{error} after the initial lateral error. . . .	88
5.26	The objective function, and the objective function after the initial lateral error. . . .	89
5.27	The gains, and the horizon.	89

6.1	Two kinds of Pareto fronts: On the left the convex kind. On the right the non-convex kind.	92
6.2	Two similar Pareto Fronts that are explored using the weighted sum scalarization. On the left: during training. On the right: Once the optimizer has converged with the crosses denoting the optimal solution found.	93
6.3	Two similar Pareto Fronts that are explored using the weighted sum scalarization. On the left: during training. On the right: Once the optimizer has converged with the crosses denoting the optimal solution found.	93
6.4	Two similar Pareto Fronts that are explored using the weighted hypervolume scalarization. On the left: during training. On the right: Once the optimizer has converged with the crosses denoting the optimal solution found.	94
6.5	Overview of the proposed method.	96
6.6	On the left: The <i>spline5</i> trajectory on a x,y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.	99
6.7	On the left: The surface corridor error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.	99
6.8	The speed over the curvilinear abscissa.	100
6.9	On the left: The <i>spline5</i> trajectory on a x,y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa (GPS loss zone in gray).	100
6.10	On the left: The surface corridor error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa (GPS loss zone in gray).	100
6.11	The speed over the curvilinear abscissa (GPS loss zone in gray).	101
6.12	The feature importance for the <i>NN controller</i> method for each input, denoted in % of importance.	101
6.13	The feature importance for the <i>Delta NN ctrl</i> method for each input, denoted in % of importance.	102
6.14	The feature importance for the <i>Model gain tuner</i> method for each input, denoted in % of importance.	102
6.15	The feature importance for the <i>Full NN gain tuner</i> method for each input, denoted in % of importance.	103
6.16	Both trajectories on a x,y global reference. On the left, the first trajectory. On the right, the second trajectory.	105
6.17	The curvature of both trajectories. On the left, the first trajectory. On the right, the second trajectory.	105
6.18	On the left: The <i>first</i> trajectory on a x,y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.	106
6.19	On the left: The surface corridor error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.	106
6.20	On the left: The speed over the curvilinear abscissa. On the right: The position accuracy over the curvilinear abscissa.	107
6.21	On the left: The <i>second</i> trajectory on a x,y scale, with the robot's path for each controller. On the right: the lateral error of the methods over the curvilinear abscissa.	107
6.22	On the left: The surface corridor error over the curvilinear abscissa. On the right: The objective function over the curvilinear abscissa.	107
6.23	On the left: The speed over the curvilinear abscissa. On the right: The position accuracy over the curvilinear abscissa.	108
7.1	Overview of the proposed method.	112
A.1	The simulated tests results over the surface error (lower is better), of a <i>NN controller</i> method (NN used for steering) optimized by each optimization method.	131
A.2	Covariance of each layer for the <i>CMA-ES</i> method. On the left the mean, on the right the standard deviation.	133
A.3	The minimum objective function found over wall time (left), and over the number of evaluations (right)	134
A.4	An example of temporal permutation, where the center input has been permuted in order to observe the change in the output.	137

A.5	Plots of the distributions: On the left a distribution plot. On the right a histogram plot	144
A.6	sine and spline0 trajectories, respectively from left to right.	145
A.7	On the left: The Adap2e robot. On the right: The trajectory over the ground.	146
A.8	The reference trajectory on an x,y scale. The trajectory at 1.0ms^{-1} for the Expert gain and the Proposed model $NN\ ob_1$. With a substantial decrease in the settling distance when comparing the Proposed model with the Expert gain.	146
A.9	In the solid lines, the predicted gain over time for the $NN\ ob_1$ method. In the dashed lines, the <i>expert constant gain</i> . In the dash-dotted lines, the errors and curvature over time.	147
A.10	left: $NN\ ob_1$ method. right: $NN\ ob_2$ method. In the solid lines, the predicted gain over time for the given method. In the dashed lines, the <i>expert constant gain</i> . In the dash-dotted lines, the errors and curvature over time.	147
A.11	Left: $NN\ ob_1$ method. Right: $NN\ ob_2$ method. In the solid lines, the predicted gain over time for the given method. In the dashed lines, the <i>expert constant gain</i> . In the dash-dotted lines, the errors, curvature, and the xy covariance (C_{xy}) over time.	147
A.12	The percent and absolute reduction of A_{error} between the method and the <i>expert constant gain</i> , in simulation.	148
A.13	The percent and absolute reduction of A_{error} between the $NN\ ob_1$ method and the <i>expert constant gain</i> , in experiment.	148
A.14	The percent and absolute reduction of A_{error} between the tested method and the <i>expert constant gain</i> , in experiment with predictive and adaptive controller.	149
A.15	Left: the percent and absolute reduction of A_{error} between the tested methods and the <i>expert constant gain</i> . Right: the percent and absolute reduction of E_{steer} between the tested methods and the <i>expert constant gain</i> , in experiment with predictive and adaptive controller, and GPS loss.	149
A.16	The trajectory, the lateral error, and the filtered lateral error. Over the total trajectory	151
A.17	The surface error A_{error} , and the surface error A_{error} after the initial lateral error.	152
A.18	The objective function, and the objective function after the initial lateral error.	152
A.19	The gains, and the horizon.	152
A.20	The trajectory, the lateral error, and the filtered lateral error. Over the total trajectory	153
A.21	The surface error A_{error} , and the surface error A_{error} after the initial lateral error.	153
A.22	The objective function, and the objective function after the initial lateral error.	154
A.23	The filtered gains, and the filtered horizon.	154
A.24	Left: the first trajectory tested. Right: the second trajectory tested.	155
A.25	Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa.	156
A.26	Left: the surface error. Center: the surface error without the initial error. Right the objective function.	156
A.27	Left: the control gains over the curvilinear abscissa. Right: the control horizon over the curvilinear abscissa.	156
A.28	Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa.	157
A.29	Left: the surface error. Center: the surface error without the initial error. Right the objective function.	157
A.30	Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa.	158
A.31	Left: the surface error. Center: the surface error without the initial error. Right the objective function.	158
A.32	Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa.	159
A.33	Left: the surface error. Center: the surface error without the initial error. Right the objective function.	159
A.34	Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa.	160
A.35	Left: the surface error. Center: the surface error without the initial error. Right the objective function.	160

A.36	Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa.	161
A.37	Left: the surface error. Center: the surface error without the initial error. Right the objective function.	161
A.38	Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa (limited to $2.0m.s^{-1}$).	162
A.39	Left: the surface error. Center: the surface error without the initial error. Right the objective function.	162
A.40	UML diagrams of the main simulation code.	165
A.41	UML diagrams of the gain tuners.	166
A.42	The <i>simulator</i> tool, without any data.	167
A.43	The <i>simulator</i> tool when valid data is given.	167
A.44	The <i>simulator</i> tool when valid and compare data is given.	168
A.45	The <i>simulator</i> tool for plotting.	168
A.46	The <i>simulator</i> tool with the real time feature importance.	168
A.47	AM ("I Have No Mouth, and I Must Scream")	169
A.48	On the left: The <i>estoril5</i> trajectory on a x,y scale. On the right: The curvature associated to this trajectory.	171
A.49	On the left: The <i>estoril7</i> trajectory on a x,y scale. On the right: The curvature associated to this trajectory.	171
A.50	On the left: The <i>estoril910</i> trajectory on a x,y scale. On the right: The curvature associated to this trajectory.	171
A.51	On the left: The <i>line</i> trajectory on a x,y scale. On the right: The curvature associated to this trajectory.	172
A.52	On the left: The <i>spline5</i> trajectory on a x,y scale. On the right: The curvature associated to this trajectory.	172
A.53	On the left: The <i>estoril6</i> trajectory on a x,y scale. On the right: The curvature associated to this trajectory.	172
A.54	On the left: The <i>estoril12</i> trajectory on a x,y scale. On the right: The curvature associated to this trajectory.	172
A.55	On the left: The <i>estoril1112</i> trajectory on a x,y scale. On the right: The curvature associated to this trajectory.	173

Appendix A

Appendices

A.1 TD Reinforcement Learning: Function derivation

Optimization target: G_t

Assuming the sum of the rewards over time describes our optimization target. Then an acceptable & computable (as the time can tend to $T \rightarrow \infty$) surrogate target would be the discounted reward over time denoted G_t , where γ is our discount factor:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots$$

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Discounting also has some intuitive sense, as it values proximal rewards at a time t :

$$1000 \text{ now} > 1000 \text{ in 1 year} > 1000 \text{ in 100 years}$$

Value function: $V(\mathfrak{s})$

In the case of reinforcement learning, ideally we want to maximize the expected return. The expected return for a given states is encoded as the *Value function*:

$$V(\mathfrak{s}) = \mathbb{E}[G_t | \mathfrak{s}, t = \mathfrak{s}]$$

From this, we can derive the following:

$$V(\mathfrak{s}) = \mathbb{E}[G_t | \mathfrak{s}, t = \mathfrak{s}]$$

$$V(\mathfrak{s}) = \mathbb{E} \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | \mathfrak{s}, t = s \right]$$

$$V(\mathfrak{s}) = \mathbb{E} \left[r_{t+1} + \gamma \left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \right) | \mathfrak{s}, t = s \right]$$

$$V(\mathfrak{s}) = \mathbb{E}[r_{t+1} + \gamma G_{t+1} | \mathfrak{s}, t = \mathfrak{s}]$$

$$V(\mathfrak{s}) = \mathbb{E}[r_{t+1} | \mathfrak{s}, t = \mathfrak{s}] + \gamma \mathbb{E}[G_{t+1} | \mathfrak{s}, t = \mathfrak{s}]$$

$$V(\mathfrak{s}) = \mathbb{E}[r_{t+1} | \mathfrak{s}, t = \mathfrak{s}] + \gamma \mathbb{E}[V(\mathfrak{s}, t+1) | \mathfrak{s}, t = \mathfrak{s}]$$

We get our final bellman equation:

$$V(\mathfrak{s}) = \mathbb{E}[r_{t+1} + \gamma V(\mathfrak{s}, t+1) | \mathfrak{s}, t = \mathfrak{s}]$$

Q-value: $Q(\mathfrak{s}, a)$

In reinforcement learning, the concept of a Q-value function is defined as the cumulative sum of the current reward r_t and future reward r_{t+1}, \dots, r_T of a given state \mathfrak{s} and action a_t .

The Q-value function is defined as such:

$$Q(\mathfrak{s}, a) = \mathbb{E} \left[\sum_{l=0}^{T-t-1} r_{t+l+1} | \mathfrak{s} \right]_{t=\mathfrak{s}, a_t=a}$$

Projecting over the optimization target G_T , the expected return for a given states and action is encoded as the *Q value*:

$$Q(\mathfrak{s}, a) = \mathbb{E}[G_t | \mathfrak{s}]_{t=\mathfrak{s}, a_t=a}$$

For $Q(\mathfrak{s}, a)$ the proof is similar to $V(\mathfrak{s})$, and we get:

$$Q(\mathfrak{s}, a) = \mathbb{E}[r_{t+1} + \gamma \max_{a'} Q(\mathfrak{s})_{t+1, a'} | \mathfrak{s}]_{t=\mathfrak{s}, a_t=a}$$

A.2 Comparing optimizer algorithms for mobile robot steering

For training the neural network in an episodic fashion, an optimizer must be used. However, there is no clear theoretical reason as to which method would allow for the best performing neural network. As such, an iterative list of incrementally complex optimizers are detailed and tested.

For the CMA-ES method, it is already detailed in section 3.4, as such it is not detailed here to avoid repetitions.

BSR: Basic Random search

When exploring a search space, an intuitive yet powerful method is stochastic search, which uses randomized information in order to explore points in a desired location. The explored locations then return information that can be used to guide the exploration location, in order to guarantee convergence towards the nearest local minimum if given enough iterations [102].

This idea is leveraged in the following optimization method called: *Basic Random Search* (*BRS*).

Algorithm 1 Basic Random Search Algorithm (from [75])

- 1: **Hyperparameters:** step-size α , number of directions sampled per iteration N , standard deviation of the exploration noise ν
- 2: **Initialize:** $\theta_0 = 0$, and $j = 0$.
- 3: **while** ending condition not satisfied **do**
- 4: Sample $\delta_1, \delta_2, \dots, \delta_N$ of the same size as θ_j , with i.i.d. standard normal entries.
- 5: Collect $2N$ rollouts of horizon H and their corresponding rewards using the policies

$$\pi_{j,k,+}(x) = \pi_{\theta_j + \nu\delta_k}(x) \quad \text{and} \quad \pi_{j,k,-}(x) = \pi_{\theta_j - \nu\delta_k}(x)$$

with $k \in 1, 2, \dots, N$.

- 6: Make the update step:

$$\theta_{j+1} = \theta_j + \frac{\alpha}{N} \sum_{k=1}^N [r(\pi_{j,k,+}) - r(\pi_{j,k,-})] \delta_k$$

- 7: $j \leftarrow j + 1$
 - 8: **end while**
-

With good starting values for the hyperparameters:

- the initial mean vector: $\alpha = 0.1$
- the exploration standard deviation vector: $\sigma = [1, \dots, 1]^T$

The *BRS* method consist of taking the current best solution θ_j , and to add and subtract a random noise vector in order to generate two candidates per random noise vector. These two candidates then return their respective objective function values, and from those an approximate direction and amplitude the solution needs to move towards can be inferred. When scaled to many random noise vector, an accurate direction and amplitude towards the local minimum can be determined.

This method is able to roughly approximate the local gradient of the exploration space using all of the sampled points, which is then used to converge toward the local minimum. Unfortunately, the exploration vector remains constant throughout the optimization process, which means that when the methods approaches the local minimum it tends to over shoot and struggle to slowly descend into the optimal point of the local minimum.

CEM: Cross-Entropy Method

A natural improvement over the *BRS* optimizer, would be to adapt the exploration to the variance in the sampled population which has the best performance. As such, one could consider a sampling

strategy that directly depends on the variance the parameters of the top e candidates obtained. This is the exact idea behind the following algorithm called: *Cross-Entropy Method (CEM)*.

Algorithm 2 Cross-Entropy Method (from [76])

- 1: **Hyperparameters:** elite size e , number of samples per iteration N , mean vector of the initial distribution μ , standard deviation vector of the initial distribution σ , .
- 2: **Initialize:** $\theta_0 = 0$, and $j = 0$.
- 3: **while** ending condition not satisfied **do**
- 4: Sample $\delta_1, \delta_2, \dots, \delta_N$ of the same size as θ_j , with the normal distribution $\mathcal{N}(\mu, \sigma^2)$.
- 5: Collect N rollouts and their corresponding objective functions using the policies

$$\pi_{\delta_k}(x)$$

with $k \in 1, 2, \dots, N$.

- 6: Sort the samples $\delta_1, \delta_2, \dots, \delta_N$ in order from lowest to highest.
- 7: Update μ from the elite:

$$\mu \leftarrow \frac{1}{e} \sum_{k=0}^e \delta_k$$

- 8: Update σ from the elite:

$$\sigma \leftarrow \sqrt{\frac{1}{e} \sum_{k=0}^e (\delta_k - \mu)^2}$$

- 9: $j \leftarrow j + 1$
 - 10: **end while**
-

With good starting values for the hyperparameters:

- elite size: $e = \max(\lceil N/5 \rceil, 1)$
- the initial mean vector: $\mu = [0, \dots, 0]^T$
- the initial standard deviation vector: $\sigma = [1, \dots, 1]^T$

Every iteration the *CEM* optimizer attempts to find the normal distribution that fits the top e candidates it sampled, and as such closes the exploration space as much as possible on the lowest point around the initial search space.

The advantage of this method, is its capability of quickly converging to the nearest local minimum with very few iterations. However, this means that it tends to not explore potentially better minimums that are further away from the initial search space (which *CMA-ES* avoids thanks to its evolution vectors and slowly decreasing exploration σ). Furthermore, by its design *CEM* ignores most of the sampled points, which is unfortunate as even points with high values contain information about where not to explore (an issue solved by the *adaptive* variation of *CMA-ES* called *aCMA-ES*).

An empirical test: Comparing with CMA-ES

With these incremental methods described and compared, an empirical analysis was done where all three methods were used to optimize a neural network which needed to steer a mobile robot from the state information in a simulation, with an objective function that minimized the lateral error and steering error (i.e. the *NN controller* method and objective function detailed in section 4.1).

The following is a distribution plot of the surface error (lower is better) returned by the neural network trained by each optimizer:

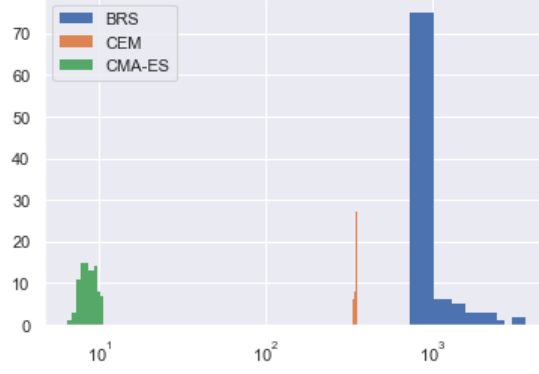


Figure A.1: The simulated tests results over the surface error (lower is better), of a *NN controller* method (NN used for steering) optimized by each optimization method.

The figure A.1 shows that the *CMA-ES* method is able to reach two orders of magnitude lower errors when compared to *BRS*, and more than one order of magnitude lower errors when compared to *CEM*. This implies that the additional elements implicit to *CMA-ES*, such as using the entire population when updating, the adaptive search space, and the evolution vectors allows it to obtain much better results.

This shows that some empirical evidence supports *CMA-ES* as the most appropriate optimizer, when compared to *CEM* and *BRS*. However, the *CMA-ES* method has many variations, as such additional tests with some variations of *CMA-ES* were also done in section A.3.

A.3 CMA-ES analysis

covariance exploration

One of the questions that was asked while using the CMA-ES method was whether or not a covariance matrix was useful for training neural networks. As a covariance matrix measures the linear covariance between two variables, and that a neural network uses activation functions to break linearity between layers, is it useful to calculate the covariance between two layers? To verify this, a neural network with a hyperbolic tangent as the activation function was trained to tune gains for a controller in a trajectory following task, and the final covariance matrix is extracted and visualized.

The parameter space for the used Neural network was of 6023 dimensions. This would be impractical to display such matrices. Furthermore, it would be visually difficult to observe any pattern with regard to the first and last layers of the neural networks, as they compose the first 880 dimensions and the last 12 dimensions respectively.

As such, the sub matrices for each weight and bias of each layer have been reduced, using the mean and standard deviation of the said sub matrices. This should give a rough overview of which parts of the parameter space, each method was converging towards.

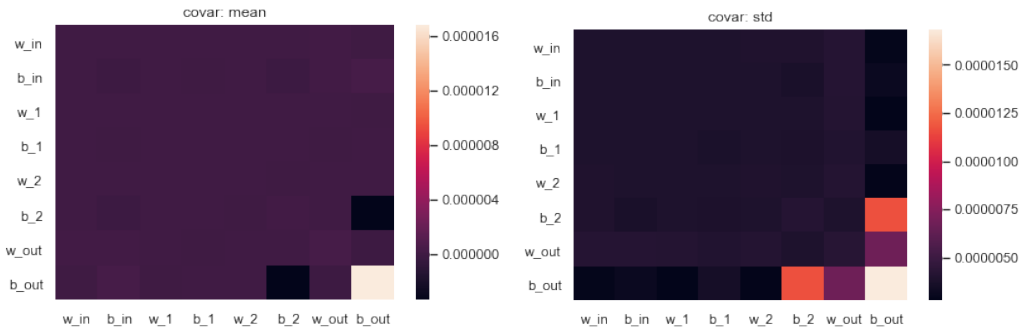


Figure A.2: Covariance of each layer for the *CMA-ES* method. On the left the mean, on the right the standard deviation.

The covariance values on figure A.2 (e.g. without the diagonal vector) of each layer for *CMA-ES*, seem to indicate a covariance of the output bias with itself, with the output weights, and with the bias of the second layer; this is expected, as the the output gains are indeed correlated with regard to the objective function (observed using a constant gain model instead of a neural network).

This implies that the full covariance matrix is not very important for neural network, as they break the linearity with their activation functions.

However calculating the covariance matrix might still be of use for the output layer, as they compute the gain from the state representation in the hidden layers.

An interesting side effect is that using a ReLU activation function ($\text{ReLU}(x) = \max(x, 0)$) might not break linearity enough, as ReLU seemed to degrade performance in the target use case. And this might be the reason why CMA-ES optimized neural networks in the state of the art tend to use other activation functions, such as hyperbolic tangent or sigmoid. However, further study is needed properly to confirm this.

CMA-ES variants

Due to its widespread use, many sub variants of the original CMA-ES method have been developed over time, each with its own advantages, disadvantages, and targeted problem space. This implies the question: Which variant of CMA-ES is optimal for the gain prediction task of a controller for a car-like mobile robot?

Most of the CMA-ES variants revolve around how the covariance matrix is calculated. Lets take as an example the original CMA-ES method with a full covariance matrix:

In order to sample from a multivariate Gaussian distribution, the covariance matrix must be square rooted:

$$\mathcal{N}(\mu, \sigma^2 C) \Leftrightarrow \mu + \sigma C^{\frac{1}{2}} \mathcal{N}(0, I)$$

Where μ is the mean, σ the global variance, and C the covariance of the search space. Calculating the square root of a matrix, is equivalent to finding the matrix A , such as:

$$A = C^{\frac{1}{2}} \Leftrightarrow AA = C$$

The most common way of doing this, is by spectral decomposition using the eigen vectors and eigen values. Finding said eigen values and eigen vectors is of $O(n^3)$ complexity. As such, the computation time is proportional to the cube of the number of dimensions in the search space, and is proportional to the population size of CMA-ES.

This is problematic when the search space is the parameter space of a neural network, as neural network commonly have 10'000 to 1'000'000 unique parameters that need optimizing. Further compounding this issue, is the question of the λ of CMA-ES, which determines how many of the population's best solutions are used to calculate the covariance matrix. In the original CMA-ES method, $\lambda = \frac{N_{\text{pop}}}{2}$. However in the adaptive variants, $\lambda = N_{\text{pop}}$, where the worst of the population is given a negative weight instead of being ignored, making CMA-ES more efficient with respect to the number of evaluations.

This means simplifying the covariance matrix to a diagonal matrix, or to a covariance matrix in a decomposed form would avoid costly calculations when large neural networks are used.

Here are some of the most common CMA-ES variants:

- *CMA-ES* [78]: uses the average of the better half of the population to compute the full covariance matrix
- *aCMA-ES* [103]: adaptive variant of CMA-ES, uses the entire population to compute the full covariance matrix
- *sepCMA-ES* [79]: separable CMA-ES, uses the average of the better half of the population to compute the diagonal of the covariance matrix, where an element wise square root is equivalent to a matrix square root.
- *sepaCMA-ES* [79, 103]: adaptive separable CMA-ES, uses the entire population to compute only the diagonal of the covariance matrix, where an element wise square root is equivalent to a matrix square root.
- *VD-CMA-ES* [104]: uses a diagonal matrix and an eigen vector, in order to compute a covariance matrix, that can be expressed as 2 vectors internally, which allows for faster computation of square root of the covariance matrix.

Each of the methods were tested in order to verify which one would be ideal for a fast computation time, while keeping the gain prediction to be as good as possible.

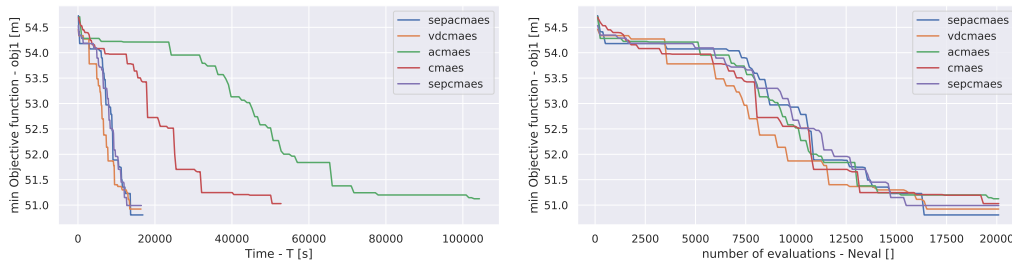


Figure A.3: The minimum objective function found over wall time (left), and over the number of evaluations (right)

Figure A.3 shows that over the number of evaluations, each method seems similar in performance. However over the all time, *sepCMA-ES*, *sepaCMA-ES*, and *VD-CMA-ES* have the clear

advantage, as they are at worst three times faster than *CMA-ES* and six times faster than *aCMA-ES*.

Nevertheless, observing training curves are not sufficient to determining the quality of the trained policy, as two similar objective function values, may have drastically different behavior. As such, 100 runs of each trained policy were run, and 4 metrics were used in order to assess the quality of the gain prediction of each policy generated by the CMA-ES methods.

$$ob_1 = \frac{1}{T} \sum_{n=0}^N \left[|y(t_n)| + L|\tilde{\theta}(t_n)| + k_{\text{steer}} L |\delta_F(t_n)| \right] \Delta t \quad [\text{m}] \quad (\text{A.1})$$

$$A_{\text{error}} = \sum_{n=0}^N \left| v(t_n) \cos(\tilde{\theta}(t_n)) \left(y(t_n) + \frac{v(t_n) \sin(\tilde{\theta}(t_n)) \Delta t}{2} \right) \right| \Delta t \quad [\text{m}^2] \quad (\text{A.2})$$

Where A_{error} is the surface of the tracking error, and ob_1 is chosen (instead of ob_2) for its lower number of parameters to tune. The notation follows the one used in the figure 2.1 in section 2.3.

When comparing over the expert method ($k_p = 0.1225$, $k_d = 0.7$) the percentage difference in performance can be observed in the table A.1.

Method	ob_1 (ms)			A_{error} (m ²)			$\int^T \tilde{\theta}_t dt$ (rad s)			$\int^T y_t dt$ (ms)		
Speed (m s ⁻¹)	1.0	1.5	2.0	1.0	1.5	2.0	1.0	1.5	2.0	1.0	1.5	2.0
CMA-ES	14%	11%	5%	41%	33%	21%	-13%	-4%	-5%	33%	26%	20%
aCMA-ES	13%	10%	5%	40%	29%	11%	-15%	-2%	6%	32%	23%	11%
VD-CMA-ES	13%	9%	5%	41%	32%	17%	-12%	-10%	-4%	33%	26%	8%
sepaCMA-ES	13%	9%	5%	42%	32%	16%	-18%	-11%	-2%	33%	26%	15%
sepCMA-ES	11%	7%	4%	32%	21%	8%	-11%	3%	6%	25%	17%	9%

Table A.1: The percent improvement over the expert method, for each metric, each speed, over each method.

This inter method similarity is even more emphasized when observing the percent improvement of each method over the expert method in the table A.1, as they do not seem to have any strong statistical outlier. Which implies that all of the tested CMA-ES methods would be comparable in terms of gain prediction quality, with the exception of *sepCMA-ES*.

This implies that the full covariance matrix is not needed to train the gain tuning method, as such if computation time becomes a large enough obstacle, then the use of *sepaCMA-ES* or *VD-CMA-ES* can be considered, as they have comparable performance to *CMA-ES*. However more research is needed, as it seems that part the covariance matrix is used to improve the training performance.

CMA-ES limitations for real world experimentation

When comparing with time difference reinforcement learning, it becomes clear that the CMA-ES reinforcement learning is far less sample efficient than the time difference reinforcement learning algorithms [71]. This means that the CMA-ES method will need far more simulated time in order to converge to a decent local optimum. In this use case, the time for the CMA-ES method is about a year of simulated time. This means that without a large number of robot in parallel [105], it would be unfeasible to train in real world condition.

As such, realistic enough simulation is needed, with possibly some transfer learning methods such as [106, 107] in order to overcome the systematic error between reality and the simulator.

A.4 New feature importance method

Feature importance

In order to explain the main variable influencing the decision of the algorithm, one can derive the *feature importance*. It analyses how important each input feature is in order to obtain a good prediction.

This is usually used in the context of decision trees [108], where each node on the tree has a score determining the quality of its split. Which in some cases is the Gini impurity [109]. The feature importance is described as the input parameters that lead to a low Gini impurity for each node that use the input feature for its split. When sorted, these feature importance show which inputs where the most useful in order to obtain a good prediction.

Unfortunately, decision trees struggle to outmatch the performance of neural networks, due to neural networks strengths as dimensional reducers and being universal function approximators for non-linear functions [9]. This means that in most cases neural networks must be used in order to obtain the desired performance.

The notion of *feature importance* is still available to neural networks, however they are not as clear as for the decision trees. The most known method is the Temporal Permutation method, described in [110], and detailed in the following section.

Temporal permutation

In [111], we described a method for determining the importance each input has with respect to the output. That is to say, how useful each input is for the neural network in order to predict the desired output. For this, the method of *temporal permutation* is used, which consist of shuffling each input over time, in order to break the coherence of that specific input, while preserving it's statistical distribution. When applying *temporal permutation* to the desired input, the observed change in the output describes the impact that input has on the prediction.

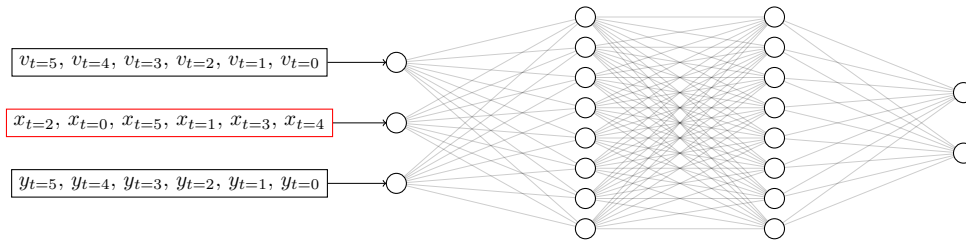


Figure A.4: An example of temporal permutation, where the center input has been permuted in order to observe the change in the output.

Figure A.4 shows a temporal permutation approach, where each input is shuffled over time in order to observe how the neural network reacts to the given input being shuffled. If the change in the neural network is strong, then that input is important to the prediction.

For a trained neural network, if the assumption that the neural network predicts a quasi-optimal output is given. Then the change between the original predicted output, and the predicted output when the input was altered, should give the influence each input has on the output. And this in turn gives describes how important each input is to predicting the quasi-optimal output.

However, this approach is not without its drawbacks, as it is dependent on any bias in the dataset used for the analysis (for instance if an input's range is not fully explored, then it might miscalculate the feature importance). Furthermore, this information is hard to exploit, as it return the expected change in the output, and not a rate of change for example.

Novel gradient base approach

As such, a novel method has been developed, consisting of exploiting the gradient of the neural network in order to determine the impact each input has for each output. This is similar to the approach used in the *Deep dream* paper [112] and with a more general approach for neural networks

than [113], as we are calculating the gradient of the output of the neural network, with respect to it's input.

Feed forward multilayer perceptron neural networks, consist of a sequence of matrix multiplications, adds, and activation functions, from the given input to the given output [9]:

$$y = a(b^{(n)} + w^{(n,n-1)}a(\dots b^{(1)} + w^{(1,0)}X)) \quad (\text{A.3})$$

Where y is the output vector, X is the input vector, a is the activation function, $b^{(n)}$ is the bias vector at the layer n , and $w^{(n,n-1)}$ is the weight matrix between the layer n and the layer $n-1$. The following shorthands are used in order to simplify the notation: $s^{(n)} = a(z^{(n)})$ and $z^{(n)} = s^{(n-1)}w^{(n,n-1)} + b^{(n)}$, where $s^{(n)}$ is the output vector of the activation function at the layer n , and $z^{(n)}$ is the vector before passing through the activation function.

From this, the gradient between the output, and any component of the neural network can be achieved using the chain rule. Indeed this is the exact method that is used in back-propagation [9] for gradient descent in supervised learning methods applied to neural networks.

$$\frac{\partial y}{\partial X} = \frac{\partial y}{\partial a} \frac{\partial a}{\partial z^{(n)}} \frac{\partial z^{(n)}}{\partial s^{(n-1)}} \dots \frac{\partial s^{(1)}}{\partial a} \frac{\partial a}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial X} \quad (\text{A.4})$$

Knowing $\frac{\partial y}{\partial a} = \frac{\partial s^{(n)}}{\partial a} = 1$, $\frac{\partial a}{\partial z^{(n)}} = a'(z^{(n)})$, $\frac{\partial z^{(n)}}{\partial s^{(n-1)}} = w^{(n,n-1)}$, and $\frac{\partial z^{(1)}}{\partial X} = w^{(1,0)}$. The following simplification is obtained using Eq. (A.4):

$$\frac{\partial y}{\partial X} = a'(z^{(n)})w^{(n,n-1)} \dots a'(z^{(1)})w^{(1,0)} \quad (\text{A.5})$$

Using the equation, we can derive the jacobian matrix between each output component and each input component. From this the expected rate of change of the output with respect to each input can be derived, and using the same assumption as the previous method, that the neural network predicts a quasi-optimal output, we can describe any change as being important to the predicted output.

Using this method we can observe the rate of change of each input with respect to each output, this implies a linearization of a given input and output set can be easily determined once the rate of change has been computed.

Deriving linear approximations

Using this mean rate of change, a linear approximation of the neural network can be inferred, and furthermore it can be derived for only a select number of inputs. This means it is possible to create a linear approximation of the previous results that depend only on the speed of the robot, using a first order Taylor approximation, derived as:

$$y \approx NN(\bar{X}) + \frac{\partial NN}{\partial X}(\bar{X})(X - \bar{X}) \quad (\text{A.6})$$

Where \bar{X} denotes the mean input of the dataset used in the generation of the mean rate of change.

Deriving N-order Taylor approximations

It was discovered that this approximation method was not limited to linear approximations, but could be used to generate any N-order multi-variable Taylor approximation.

$$\begin{aligned} NN(x_1, x_2, \dots, x_d) = & NN(a_1, a_2, \dots, a_d) + \sum_{j=1}^d \frac{\partial NN(a_1, a_2, \dots, a_d)}{\partial x_j} (x_j - a_j) \\ & + \frac{1}{2!} \sum_{j=1}^d \sum_{k=1}^d \frac{\partial^2 NN(a_1, a_2, \dots, a_d)}{\partial x_j \partial x_k} (x_j - a_j)(x_k - a_k) \\ & + \frac{1}{3!} \sum_{j=1}^d \sum_{k=1}^d \sum_{l=1}^d \frac{\partial^3 NN(a_1, a_2, \dots, a_d)}{\partial x_j \partial x_k \partial x_l} (x_j - a_j)(x_k - a_k)(x_l - a_l) + \dots \end{aligned} \quad (\text{A.7})$$

This could in turn allow for approximate, but interpretable and predictable behavior of a neural network, allowing for proofs of stability, or simplifications for faster calculations. However, it should be noted that an N-order approximation requires up to $\sum_{k=0}^N d^k = \frac{d^{N+1}-1}{d-1}$ unique multiplications

(if the partial derivatives are precomputed), which can be considerably higher than computing the neural network. For example with $d = 20$ inputs, an 4th order approximation requires up to 8421 unique multiplications.

However, this method suffers from the choice of activation function. Indeed for a valid Taylor approximation, the neural network must be infinitely differentiable, this only occurs if the activation function is infinitely differentiable. This means that activation functions such as ReLU cannot be used with this method.

Furthermore in order to get the ideal performance, a well behaved function must be used. A well behaved function is one where higher order derivatives have a lower impact on the approximation, as in their Nth gradient factor does not grow as fast as the factorial of N. As such, if a ill behaved activation function is used, then the higher order derivatives of the activation function will cause a degradation on the generalization outside of the approximation point.

Examples of ill behaved functions include but are not limited to: gaussian, tanh, LiSHT, Bent identity, & ArcTan.

Example of well behaved functions include but are not limited to: sigmoid, sigmoid linear unit (SiLU), softplus, gaussian error linear unit, sin, & sinc.

Gradient base feature importance of experimental results

Using this method, an analysis of the previous results can be done, and from this analysis a better understanding of the importance of the inputs can be ascertained. By dividing each mean rate of change by the expected value and normalizing the results, a graph with percentage of contribution each input has can be derived. This allows for problem and robot specific insights to be ascertained after training of the neural network, with allows for a better understanding of the neural network's behavior in any given situation.

A.5 NN controller and Delta NN ctrl with dynamic parameters

Due to the poor initial results of *NN controller* and *Delta NN ctrl* in section 4.1, one could consider excluding them from the section 5. However, due to the additional inputs suggested by section 5, tests must be done to validate that the performance of *NN controller* and *Delta NN ctrl* are still below the *NN gain tuner* method with the additional inputs. For this, the following tables show the simulate results over the testing trajectories and speeds of a Full *NN controller* method and Full *Delta NN ctrl* method, where the inputs are identical to the *Full NN gain tuner* method:

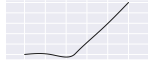



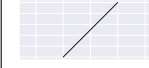
						
		estoril5	estoril7	estoril910	spline5	line
$1m.s^{-1}$	Romea	1.44 (± 0.33)	2.08 (± 0.64)	29.85 (± 0.90)	3.93 (± 1.01)	1.09 (± 0.04)
	<i>Model gain tuner</i>	1.14 (± 0.22)	1.29 (± 0.54)	25.94 (± 1.17)	2.67 (± 1.15)	1.09 (± 0.03)
	Full <i>NN controller</i>	2.20 (± 0.38)	2.57 (± 0.63)	21.85 (± 1.17)	4.14 (± 1.57)	2.16 (± 0.35)
	Full <i>Delta NN ctrl</i>	1.24 (± 0.19)	1.29 (± 0.47)	24.93 (± 1.15)	2.17 (± 1.00)	1.15 (± 0.05)
	<i>Full NN gain tuner</i>	1.17 (± 0.21)	1.28 (± 0.51)	21.02 (± 1.52)	1.97 (± 0.89)	1.17 (± 0.03)
	<i>NN gain tuner</i>	1.26 (± 0.23)	1.60 (± 0.57)	19.82 (± 1.92)	2.30 (± 1.19)	1.17 (± 0.03)
$2m.s^{-1}$	Romea	2.04 (± 0.63)	3.65 (± 1.00)	34.79 (± 1.79)	6.67 (± 1.16)	1.13 (± 0.06)
	<i>Model gain tuner</i>	1.69 (± 0.72)	2.67 (± 1.29)	31.63 (± 2.74)	4.98 (± 1.68)	1.16 (± 0.06)
	Full <i>NN controller</i>	2.85 (± 0.46)	3.58 (± 0.95)	18.49 (± 1.36)	5.46 (± 1.36)	2.00 (± 0.22)
	Full <i>Delta NN ctrl</i>	2.49 (± 0.38)	3.54 (± 0.61)	19.12 (± 1.89)	5.10 (± 1.06)	1.52 (± 0.20)
	<i>Full NN gain tuner</i>	1.67 (± 0.41)	2.19 (± 0.94)	17.68 (± 2.09)	3.62 (± 1.23)	1.36 (± 0.07)
	<i>NN gain tuner</i>	1.67 (± 0.39)	2.38 (± 0.97)	17.45 (± 1.79)	3.66 (± 1.47)	1.35 (± 0.08)
$3m.s^{-1}$	Romea	4.23 (± 1.22)	8.20 (± 1.59)	56.66 (± 5.16)	12.94 (± 1.89)	1.22 (± 0.12)
	<i>Model gain tuner</i>	2.83 (± 1.82)	4.75 (± 3.08)	44.25 (± 9.10)	7.98 (± 3.44)	1.33 (± 0.12)
	Full <i>NN controller</i>	3.87 (± 0.62)	4.91 (± 0.87)	20.93 (± 2.02)	7.13 (± 1.66)	2.77 (± 0.32)
	Full <i>Delta NN ctrl</i>	3.39 (± 0.60)	5.02 (± 0.89)	19.86 (± 2.48)	7.84 (± 1.46)	1.64 (± 0.20)
	<i>Full NN gain tuner</i>	2.20 (± 0.81)	3.28 (± 1.55)	19.06 (± 3.46)	5.88 (± 2.17)	1.42 (± 0.10)
	<i>NN gain tuner</i>	2.41 (± 0.69)	3.50 (± 1.34)	20.55 (± 4.16)	7.24 (± 2.85)	1.50 (± 0.14)
$4m.s^{-1}$	Romea	5.48 (± 1.57)	10.37 (± 2.36)	60.78 (± 9.40)	15.72 (± 3.82)	1.26 (± 0.15)
	<i>Model gain tuner</i>	4.94 (± 3.88)	8.00 (± 5.57)	70.99 (± 24.00)	14.05 (± 6.57)	1.61 (± 0.25)
	Full <i>NN controller</i>	6.75 (± 1.36)	8.09 (± 1.52)	33.52 (± 6.78)	11.73 (± 5.31)	5.40 (± 0.97)
	Full <i>Delta NN ctrl</i>	4.81 (± 1.24)	6.47 (± 1.21)	29.15 (± 3.80)	9.89 (± 2.35)	2.60 (± 0.49)
	<i>Full NN gain tuner</i>	3.64 (± 2.37)	5.19 (± 2.68)	28.61 (± 20.18)	9.33 (± 4.46)	1.70 (± 0.17)
	<i>NN gain tuner</i>	4.03 (± 2.22)	5.52 (± 2.04)	28.02 (± 6.38)	12.66 (± 6.88)	1.81 (± 0.24)

Table A.2: Surface error in [m2] of each method at all the speeds and trajectories used during training, with an **initial error** of $0m$

Figure A.2 shows the performance of Full *NN controller* and Full *Delta NN ctrl*, which seem overall to have lower performance when compared to *NN gain tuner*, with Full *NN controller* having the lowest of the three methods. Furthermore, it seems that adding the additional inputs has not improved the *Delta NN ctrl* method and has in fact degraded the performance of *NN controller* (as can be compared with table 4.7 in section 4.3).

When comparing Full *NN controller* and Full *Delta NN ctrl* with *Full NN gain tuner* the results is clear, as *Full NN gain tuner* is able to outperform Full *NN controller* and Full *Delta NN ctrl* in every trajectory and at every speed tested.

Figure A.3 shows similar results as the previous table, where overall *Full NN gain tuner* outperforms Full *Delta NN ctrl* and Full *NN controller*, with a slight performance gap over *spline5* at $4m.s^{-1}$ for the Full *Delta NN ctrl* method.

Overall, it seems that adding the additional inputs to the *Delta NN ctrl* has only slightly change the performance, but remains below *Full NN gain tuner*, and the performance of Full *NN controller* has dropped with respect to *NN controller*. This shows that the exclusion of these methods in the section 5 are indeed valid.

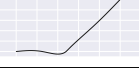

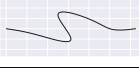


						
		estoril5	estoril7	estoril910	spline5	line
$1m.s^{-1}$	Romea	4.70 (± 0.33)	5.33 (± 0.63)	33.12 (± 0.91)	7.15 (± 1.01)	4.35 (± 0.08)
	<i>Model gain tuner</i>	2.92 (± 0.26)	3.11 (± 0.53)	27.76 (± 1.20)	4.38 (± 1.17)	2.89 (± 0.12)
	Full <i>NN controller</i>	4.77 (± 0.35)	5.14 (± 0.63)	24.45 (± 1.15)	6.67 (± 1.57)	4.74 (± 0.30)
	Full <i>Delta NN ctrl</i>	3.07 (± 0.22)	3.14 (± 0.47)	26.74 (± 1.11)	3.96 (± 1.02)	2.98 (± 0.12)
	<i>Full NN gain tuner</i>	2.98 (± 0.22)	3.09 (± 0.51)	22.86 (± 1.54)	3.80 (± 0.86)	2.97 (± 0.09)
	<i>NN gain tuner</i>	3.08 (± 0.24)	3.44 (± 0.56)	21.64 (± 1.93)	4.20 (± 1.14)	3.00 (± 0.08)
$2m.s^{-1}$	Romea	6.50 (± 0.61)	8.06 (± 0.99)	39.17 (± 1.79)	10.82 (± 1.16)	5.52 (± 0.10)
	<i>Model gain tuner</i>	3.53 (± 0.72)	4.46 (± 1.30)	33.44 (± 2.83)	6.68 (± 1.69)	2.99 (± 0.22)
	Full <i>NN controller</i>	5.16 (± 0.44)	5.87 (± 0.96)	20.80 (± 1.37)	7.78 (± 1.38)	4.36 (± 0.21)
	Full <i>Delta NN ctrl</i>	4.41 (± 0.38)	5.45 (± 0.58)	21.00 (± 1.85)	6.92 (± 1.06)	3.40 (± 0.20)
	<i>Full NN gain tuner</i>	3.38 (± 0.41)	3.88 (± 0.94)	19.42 (± 2.08)	5.18 (± 1.34)	3.11 (± 0.14)
	<i>NN gain tuner</i>	3.36 (± 0.40)	4.07 (± 0.98)	19.11 (± 1.82)	5.44 (± 1.64)	3.09 (± 0.15)
$3m.s^{-1}$	Romea	12.31 (± 1.22)	16.09 (± 1.58)	64.85 (± 5.14)	19.94 (± 1.95)	9.21 (± 0.15)
	<i>Model gain tuner</i>	4.82 (± 1.83)	6.78 (± 3.05)	46.26 (± 9.07)	9.94 (± 3.36)	3.32 (± 0.44)
	Full <i>NN controller</i>	6.22 (± 0.57)	7.19 (± 0.88)	23.30 (± 2.03)	9.56 (± 1.59)	5.12 (± 0.30)
	Full <i>Delta NN ctrl</i>	5.35 (± 0.61)	7.09 (± 0.87)	21.87 (± 2.46)	9.82 (± 1.48)	3.62 (± 0.26)
	<i>Full NN gain tuner</i>	3.97 (± 0.81)	5.01 (± 1.54)	20.81 (± 3.44)	7.99 (± 2.32)	3.27 (± 0.23)
	<i>NN gain tuner</i>	4.08 (± 0.76)	5.18 (± 1.40)	22.41 (± 4.27)	9.20 (± 2.93)	3.26 (± 0.28)
$4m.s^{-1}$	Romea	13.44 (± 1.59)	18.13 (± 2.37)	68.75 (± 9.39)	22.68 (± 3.91)	9.13 (± 0.17)
	<i>Model gain tuner</i>	6.96 (± 3.94)	10.02 (± 5.53)	73.06 (± 24.10)	16.13 (± 6.57)	3.66 (± 0.72)
	Full <i>NN controller</i>	9.12 (± 1.31)	10.51 (± 2.00)	35.97 (± 6.98)	13.99 (± 5.23)	7.83 (± 1.01)
	Full <i>Delta NN ctrl</i>	6.76 (± 1.25)	8.44 (± 1.17)	31.14 (± 3.74)	11.79 (± 2.18)	4.62 (± 0.54)
	<i>Full NN gain tuner</i>	5.41 (± 2.41)	6.93 (± 2.52)	30.09 (± 12.95)	12.02 (± 18.20)	3.59 (± 0.30)
	<i>NN gain tuner</i>	5.65 (± 2.17)	7.25 (± 2.10)	29.44 (± 6.45)	14.26 (± 6.65)	3.61 (± 0.43)

Table A.3: Surface error in [m2] of each method at all the speeds and trajectories used during training, with an **initial error of 1m**

A.6 Training method variance test: *Full NN gain tuner* case study

In order to justify the methods used, a variance test was performed on the *Full NN gain tuner*, in order to validate that the model obtained was close to the statistical mean, and is not an outlier. Furthermore, this test is done to show the variability of the training method that is used, with respect to the performance obtained.

<i>Full NN gain tuner</i>	run 1	run 2	run 3	run 4	run 5
mean	7.63144	7.78528	7.52629	7.39438	7.6859
standard deviation	7.97336	7.58792	6.65082	6.44512	6.96599

Table A.4: The overall Surface error in $[m^2]$ of each run of *Full NN gain tuner* at all the speeds and trajectories used during training, with an **initial error of 1m**.

Table A.4 shows the mean and standard deviation of the surface error for each run overall. These results imply an overall mean of 7.60466 and a standard deviation of 0.134388 (with a 5.02% between min and max values), which means a standard deviation of 1.77% can be expected in practice, and as such any differences between results that are below 3,54% (2σ) can be seen as variance of the training method.

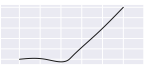




						
		estoril5	estoril7	estoril910	spline5	line
$1m.s^{-1}$	<i>Full NN gain tuner</i>	2.98 (± 0.21)	3.12 (± 0.54)	22.76 (± 1.32)	3.87 (± 0.84)	2.98 (± 0.10)
	<i>Full NN gain tuner</i> (2)	3.03 (± 0.23)	3.32 (± 0.48)	21.98 (± 1.69)	3.86 (± 0.96)	3.00 (± 0.14)
	<i>Full NN gain tuner</i> (3)	2.97 (± 0.25)	3.21 (± 0.61)	21.66 (± 1.50)	3.75 (± 0.89)	2.97 (± 0.14)
	<i>Full NN gain tuner</i> (4)	2.93 (± 0.22)	3.05 (± 0.55)	23.33 (± 1.20)	3.68 (± 0.88)	2.96 (± 0.11)
	<i>Full NN gain tuner</i> (5)	2.94 (± 0.22)	3.06 (± 0.48)	24.85 (± 1.00)	3.79 (± 0.75)	2.99 (± 0.10)
$2m.s^{-1}$	<i>Full NN gain tuner</i>	3.38 (± 0.39)	3.78 (± 0.91)	19.33 (± 1.99)	5.32 (± 1.61)	3.10 (± 0.12)
	<i>Full NN gain tuner</i> (2)	3.61 (± 0.40)	4.01 (± 0.69)	18.96 (± 2.28)	5.40 (± 2.97)	3.12 (± 0.12)
	<i>Full NN gain tuner</i> (3)	3.53 (± 0.46)	4.17 (± 0.93)	20.16 (± 1.79)	6.26 (± 3.94)	3.15 (± 0.12)
	<i>Full NN gain tuner</i> (4)	3.42 (± 0.47)	3.94 (± 0.88)	19.47 (± 1.87)	5.90 (± 3.12)	3.07 (± 0.12)
	<i>Full NN gain tuner</i> (5)	3.34 (± 0.49)	3.76 (± 0.87)	22.20 (± 2.51)	5.63 (± 3.07)	3.19 (± 0.21)
$3m.s^{-1}$	<i>Full NN gain tuner</i>	3.97 (± 0.85)	4.89 (± 1.49)	20.68 (± 3.71)	8.11 (± 2.23)	3.26 (± 0.19)
	<i>Full NN gain tuner</i> (2)	4.30 (± 0.73)	5.00 (± 1.12)	21.14 (± 3.35)	7.60 (± 3.48)	3.22 (± 0.19)
	<i>Full NN gain tuner</i> (3)	4.22 (± 0.97)	5.40 (± 1.30)	21.89 (± 3.39)	8.12 (± 3.00)	3.34 (± 0.23)
	<i>Full NN gain tuner</i> (4)	4.39 (± 0.87)	5.18 (± 1.38)	21.13 (± 3.36)	8.33 (± 4.66)	3.27 (± 0.21)
	<i>Full NN gain tuner</i> (5)	4.40 (± 1.05)	5.31 (± 1.25)	22.49 (± 3.95)	8.54 (± 3.23)	3.52 (± 0.44)
$4m.s^{-1}$	<i>Full NN gain tuner</i>	5.33 (± 2.41)	6.62 (± 2.44)	28.95 (± 7.88)	11.72 (± 4.88)	3.59 (± 0.30)
	<i>Full NN gain tuner</i> (2)	5.78 (± 2.18)	6.65 (± 2.02)	30.16 (± 12.71)	13.10 (± 7.88)	3.58 (± 0.28)
	<i>Full NN gain tuner</i> (3)	5.55 (± 2.07)	7.12 (± 2.37)	29.35 (± 10.53)	12.35 (± 8.04)	3.85 (± 0.47)
	<i>Full NN gain tuner</i> (4)	5.86 (± 1.53)	7.04 (± 2.56)	28.16 (± 7.08)	11.69 (± 5.17)	3.65 (± 0.43)
	<i>Full NN gain tuner</i> (5)	6.02 (± 2.09)	7.07 (± 2.44)	28.42 (± 5.19)	12.33 (± 6.04)	4.14 (± 0.84)

Table A.5: Surface error in $[m^2]$ of each method at all the speeds and trajectories used during training, with an **initial error of 1m**.

When comparing the runs over the trajectories and speeds on table A.5, it seems that the variances might be subtle changes in the control policy, as some runs perform better than other in some cases. But overall no clear better method can be distinguished.

Figure A.5 shows the distribution and histogram of the each run over the results obtained in simulation. It shows that they all have similar distribution profiles with very little skew, and with a consistent histogram (with the exception of the 5th run). This shows that the runs overall generate very similar results.

Overall, the results have shown that the variance of the training methods is quite small, and the results obtained can be very similar, with an overall variation of 1.77%.

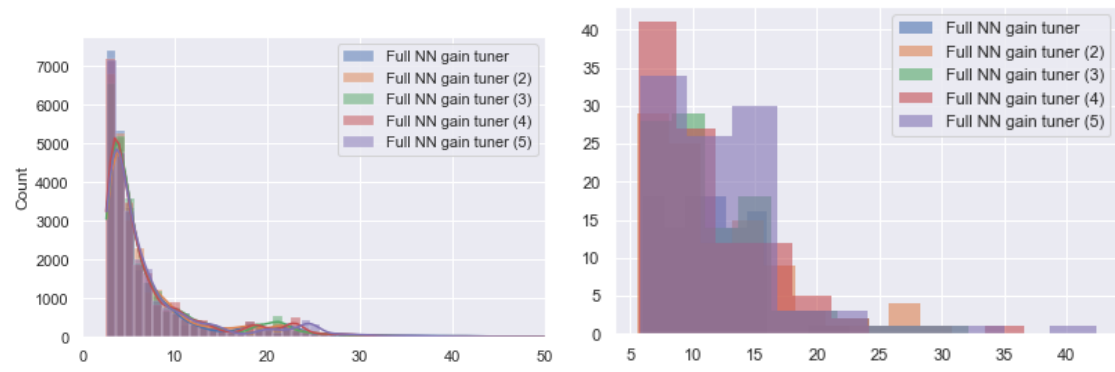


Figure A.5: Plots of the distributions: On the left a distribution plot. On the right a histogram plot

A.7 Gain synthesis using CMA-ES in dynamic simulation

In this Annex we detail further some of the first simulated and experimental results obtained during the PhD for gain adaptation using CMA-ES.

Gain synthesis method being validated in a kinematic simulator, it now needs to be tried in real world testing. For this, a dynamic model of the mobile robot is used for the training in simulation, with actuator delays and sliding dynamics, as it more closely models reality. However, this means that the method must be tuned for a very specific robot and environment.

Experimental setup

Training configuration

The CMA-ES method is run with an initial standard deviation of $\sigma = 0.04$, an initial average of $\mu = 0$, a population size of 32 (from [78], $N_{\text{pop}} = 4 + \lfloor 3 \log(N) \rfloor$, with $N = 6000$, $N_{\text{pop}} = 30$, rounded up to 32 for CPU core counts), and a maximum of 20000 evaluations before termination.

It uses the simulator detailed in section 2.8, based on the dynamical model detailed in section (2.3), fed with Pacejka contact forces (2.3). If this permits to investigate the influence of sliding, the Pacejka model has first been set as constant to be representative of the motion on a gravel road (intermediate grip conditions).

The neural network is of approximately 6000 parameters, activation function is hyperbolic tangent, with hidden layers of 40, 100, and 10 neurons respectively. The neural network takes as inputs the lateral error, angular error, GPS position covariance, current curvature, future curvature (10 samples over 1s), and the derivative of the lateral and angular error. The output is 2 gains: k_p and k_d .

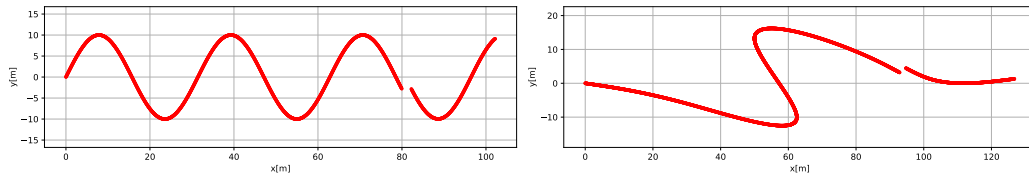


Figure A.6: sine and spline0 trajectories, respectively from left to right.

The evaluation for neural network weights was done over the 2 trajectories shown in the figure A.6, and over the speeds of 1.0m s^{-1} , 1.5m s^{-1} and 2.0m s^{-1} , 5 times each to reduce the variance in the evaluation and to verify the stability of the controller with the dynamic gain. At a random location on each trajectory, a simulated loss of GPS signal is done, where the accuracy of the position sensor goes from 0.01m, to 1.0m.

The control law used is Romea for training and evaluation, and the Romea predictive one for evaluation, With an Extended Kalman Filter (EKF) for the state estimation, and a Lyapunov sliding observer.

Two models were trained for these experiments. The first one is called $NN\ ob_1$. It is trained using the objective function ob_1 with a $k_{\text{steer}} = 0.5$. The second one is called $NN\ ob_2$. It is trained using the objective function ob_2 with a $\lambda = 6.0$. Several values of λ were tested in simulation ($\lambda = 1$, $\lambda = 6$, and $\lambda = 12$), in order to select the best one that would dampen oscillations, while correcting the errors.

Experimentation in simulation

The trained neural network is then validated in simulation and compared to a gain model tuned by an expert, where the gain is set to $k_p = 0.7$, and $k_d = 0.1225$. Each validation is done over the two trajectories shown in the figure A.6, and over the speeds 1.0m s^{-1} , 1.5m s^{-1} , and 2.0m s^{-1} .

Experimentation with the Adap2e platform

After the validation of the simulated results, the gain prediction methods is tested with a robot in real world conditions, at a speed of 1.0m s^{-1} , 1.5m s^{-1} , and 2.0m s^{-1} .



Figure A.7: On the left: The Adap2e robot. On the right: The trajectory over the ground.

The robot used is the Adap2e platform (figure A.7). This platform has a wheel base length of 1.38m, a wheel track length of 1.0m, a mass of 400kg, a steering actuation delay of about 0.4s, a GPS position update rate of 10hz, control loop update rate of 10hz, and a settling time for the steering angle in 0.5s.

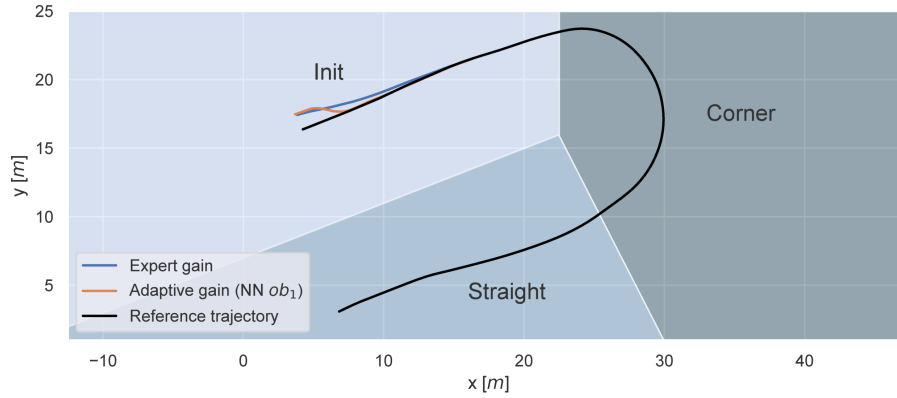


Figure A.8: The reference trajectory on an x,y scale. The trajectory at 1.0ms^{-1} for the Expert gain and the Proposed model $NN\ ob_1$. With a substantial decrease in the settling distance when comparing the Proposed model with the Expert gain.

The trajectory is segmented into three parts shown on the figure A.8:

- *Init*: The start of the trajectory, the robot is launched from approximately 1m from the side of the trajectory as an initial error.
- *Corner*: The large corner of the trajectory, with a constant curvature of 0.2m^{-1} .
- *Straight*: The final straight line after the large corner, in order to observe the stabilization from the corner.

Null hypothesis

In this experiment, the following null hypothesis must be refuted:

- *Adjusting the gain in real time, will yield similar or worse performance to using a constant gain.*

Should the null hypothesis be refuted, then it would mean adjusting the gain in real time, would allow for better performance than using a constant gain, as it is able to adapt in real time to the perceived changes in the environmental state, while being robust to the sliding dynamics and actuation delay.

Results

Qualitative results

Here in the qualitative results, the interpretation of the gain from the proposed method will be done, in order to see the adaptation of each objective function, relative to the observation.

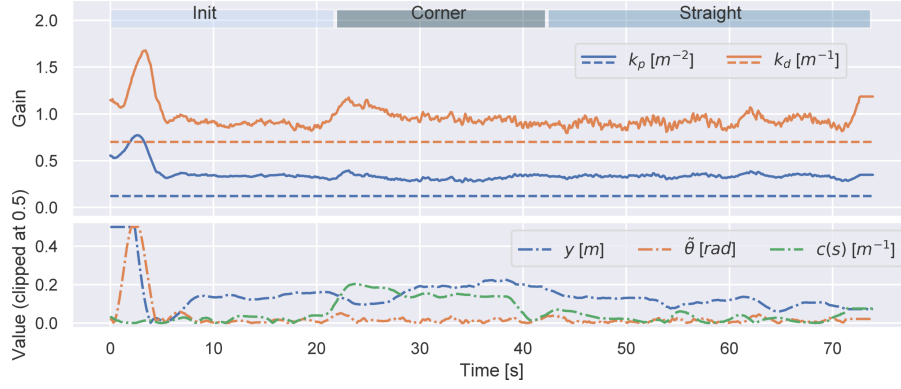


Figure A.9: In the solid lines, the predicted gain over time for the $NN ob_1$ method. In the dashed lines, the *expert constant gain*. In the dash-dotted lines, the errors and curvature over time.

The first test is run with the Romea classic controller without sliding accounted. The gain output with the $NN ob_1$ gain prediction model, at 1.0ms^{-1} can be seen on the figure A.9. A large spike at the *Init* section, shows that the method is correcting for the large initial error quickly without overshooting. Then, a small increase in the gain occurs at the *Corner* section in order to follow the curvature. One can also see the effect of neglecting sideslip angles, since the robot does not perfectly match the reference path (see the lateral error in figure A.9).

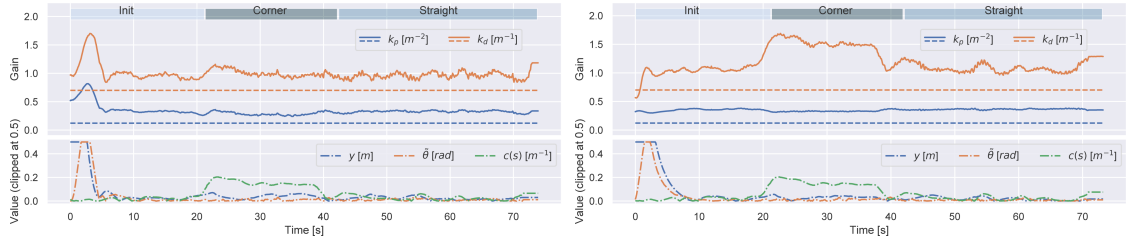


Figure A.10: left: $NN ob_1$ method. right: $NN ob_2$ method. In the solid lines, the predicted gain over time for the given method. In the dashed lines, the *expert constant gain*. In the dash-dotted lines, the errors and curvature over time.

The next test is run with the Romea predictive controller with sliding accounted. The gain output at 1.0ms^{-1} can be seen on the figure A.10. A large spike at the *Init* section, shows that the method is correcting for the large initial error quickly without overshooting for $NN ob_1$. Then, a small increase in the gain at the *Corner* section for $NN ob_1$, and a large increase for $NN ob_2$, occur in order to follow the curvature. Taking into account sliding makes it possible to better follow the reference path.

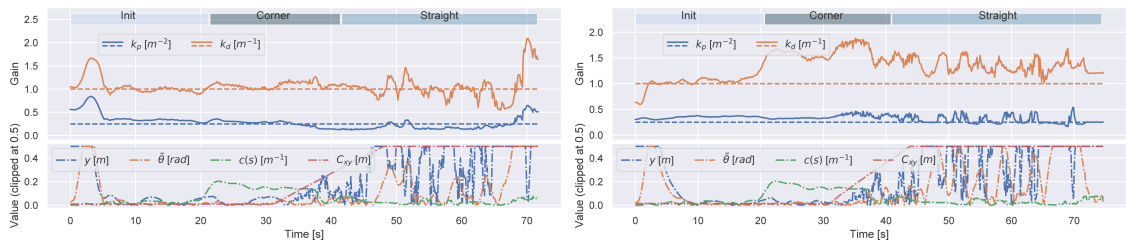


Figure A.11: Left: $NN ob_1$ method. Right: $NN ob_2$ method. In the solid lines, the predicted gain over time for the given method. In the dashed lines, the *expert constant gain*. In the dash-dotted lines, the errors, curvature, and the xy covariance (C_{xy}) over time.

The final test is run with the Romea predictive controller, with a GPS signal loss after 30m. The gain output at 1.0ms^{-1} can be seen on the figure A.11. Identical behavior to the previous experiment can be observed for the two first sections. The final section shows a drop in the k_d

gain for the $NN\ ob_1$ method, which implies a higher damping ratio, reducing the reactivity of the controller due to the GPS noise.

Quantitative results

In machine learning, when a metric becomes a target, it can cause the metric to become useless, as the training method will overfit to that metric at the detriment of other metrics.

For this, the following metrics will be used:

$$A_{\text{error}} = \sum_{n=0}^N \left| v(t_n) \cos(\tilde{\theta}(t_n)) \left(y(t_n) + \frac{v(t_n) \sin(\tilde{\theta}(t_n)) \Delta t}{2} \right) \right| \Delta t \quad [\text{m}^2] \quad (\text{A.8})$$

$$E_{\text{steer}} = \sum_{n=0}^N |\delta_F(t_n)| \Delta t \quad [\text{rad s}] \quad (\text{A.9})$$

Where $\tilde{\theta}$ is the angular error relative to the trajectory, v is the speed, y is the lateral error relative to the trajectory, and δ_F is the front steering. This allows the surface of the error A_{error} and the steering energy E_{steer} to be measured and evaluated, while keeping the objective functions ob_1 and ob_2 for training.

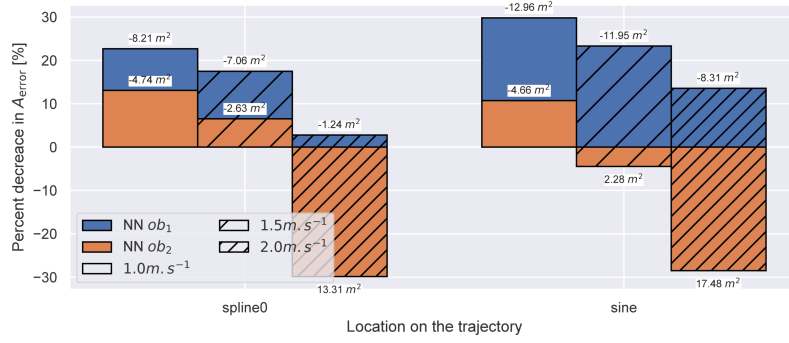


Figure A.12: The percent and absolute reduction of A_{error} between the method and the *expert constant gain*, in simulation.

The simulated results on figure A.12, show the $NN\ ob_1$ gain prediction model is capable of following the trajectory more closely than the expert gain model in the trained trajectories. $NN\ ob_2$ however, shows a sharp reduction in performance with a speed higher than 1.5m.s^{-1} .

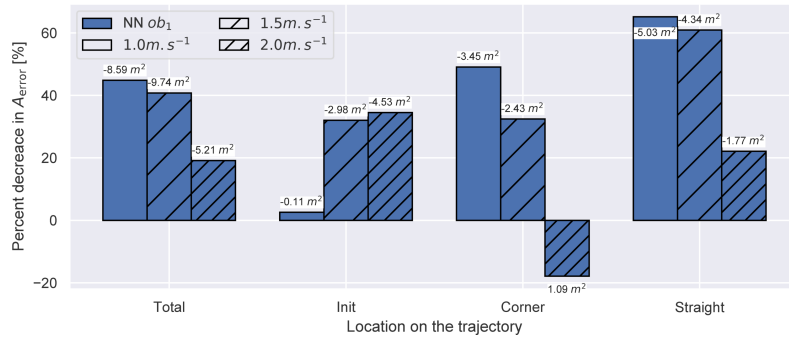


Figure A.13: The percent and absolute reduction of A_{error} between the $NN\ ob_1$ method and the *expert constant gain*, in experiment.

The method being validated in simulation, it can now be tested in real world conditions with the Adap2e platform. The results from the first experiment with the Romea classic controller and $NN\ ob_1$ gain prediction model can be seen on figure A.13. Here, an overall decrease in the error of at least 20% can be observed, with a specific drop in performance for the *Corner* section at 2.0m.s^{-1} , likely due to the systemic error at higher speeds.

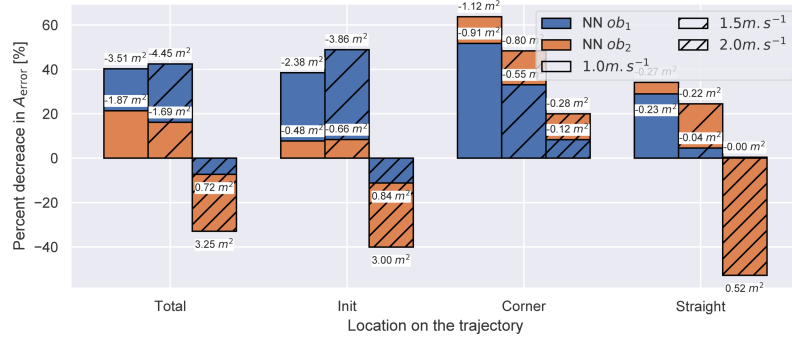


Figure A.14: The percent and absolute reduction of A_{error} between the tested method and the *expert constant gain*, in experiment with predictive and adaptive controller.

The next experiment consists of testing the $NN\ ob_1$ and $NN\ ob_2$ gain prediction models, on the Adap2e platform, with the Romea predictive controller and Lyapunov sliding observer. The results of the experiment can be seen on the figure A.14. Here, similar performance to the previous experiment can be observed, with the exception of the 2.0m.s^{-1} speed, where a higher error occurs. This is expected, as the methods were trained for the Romea classic controller, and the higher dynamic effects start to become apparent at 2.0m.s^{-1} . $NN\ ob_2$ shows similar performance to those seen in the simulated results, signifying that ob_2 may not be properly tuned for the task.

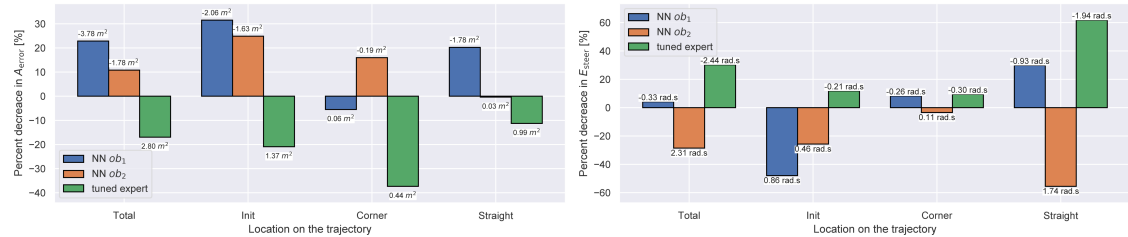


Figure A.15: Left: the percent and absolute reduction of A_{error} between the tested methods and the *expert constant gain*. Right: the percent and absolute reduction of E_{steer} between the tested methods and the *expert constant gain*, in experiment with predictive and adaptive controller, and GPS loss.

The final experiment consists of testing the $NN\ ob_1$ and $NN\ ob_2$ gain prediction models, on the Adap2e platform, with the Romea predictive controller and Lyapunov sliding observer, with a GPS noise of about 1.0m after 30m at a speed of 1.0m.s^{-1} for safety reasons. The results of the experiment can be seen on the figure A.15, where the expert gain is set to $k_p = 1.0$ and $k_d = 0.25$ to reflect the lowered constant speed. Here, $NN\ ob_1$ and $NN\ ob_2$ show a 20% and a 10% decrease in the surface error respectively, while keeping similar steering energy for $NN\ ob_1$. The tuned gains are set to $k_p = 0.7$ and $k_d = 0.1225$, and show an increase in the surface error.

Overall, the gain prediction models show an adaptation of the gain, relative to the changes in speed, curvature, errors, and Kalman covariance. This adaptation allows for up to 40% reduction in the overall surface error, which allows the rejection of the null hypothesis. Furthermore, this adaptation is without any transfer learning, which shows that it may not be necessary, if the environment can be modeled accurately enough.

It should be noted that $NN\ ob_2$'s poor performance might be due to badly tuned parameters (such as the frequencies or λ), or a missing component to the loss function. These are aspect currently being tested.

Limitations

There are some limitations that are visible after the experiments. The first is the gain prediction model, as it is not taking into account the sliding angles, from the sliding observes to determine the gain.

The second is the training took place with fixed sliding conditions, where as in reality, the sliding condition could change drastically over time. As such the training process should change the sliding conditions over the evaluation.

A.8 Real world trials with dynamic parameters

Section 5.3 shows selected experimental results, representative of the testing campaign. In this annex further experimental cases are detailed.

the following results were obtained with the objective function:

$$ob_3 = \frac{1}{T} \sum_{n=0}^N [|y(t_n)| + k_{steer} L |\delta_F(t_n)|] \Delta t \quad [\text{m}] \quad (\text{A.10})$$

Furthermore, the analysis of the results is performed with the surface error:

$$A_{\text{error}} = \sum_{n=0}^N \left| v(t_n) \cos(\tilde{\theta}(t_n)) \left(y(t_n) + \frac{v(t_n) \sin(\tilde{\theta}(t_n)) \Delta t}{2} \right) \right| \Delta t \quad [\text{m}^2] \quad (\text{A.11})$$

This is done, in order to validate the performance of the trained neural network, without resorting to the objective function. Indeed, when a reinforcement learning agent trains to optimize a function, it is possible that the said agent might exploit the objective function in order to minimize it, without achieving the desired behavior. For the following, experiments are achieved with RobuFAST, in the same conditions that are detailed in section 5.3

Trials with the Pacejka tyre model for training

The neural network, once trained in a dynamic simulation with the Pacejka tyre model, was implemented along with the model based gain tuning method, on the RobuFAST platform for testing at 4m.s^{-1} :

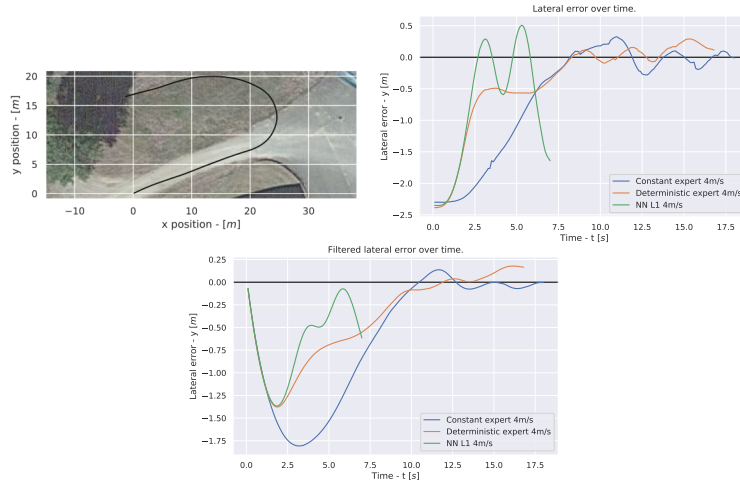


Figure A.16: The trajectory, the lateral error, and the filtered lateral error. Over the total trajectory

From the Fig A.16, we can see the error over the entire trajectory is considerably lower with the model based gain tuning method. On the contrary the proposed NN method showed very dangerous instability when started (hence the oscillations seen above), as such the experiment needed to be interrupted for safety reasons for the proposed NN method.

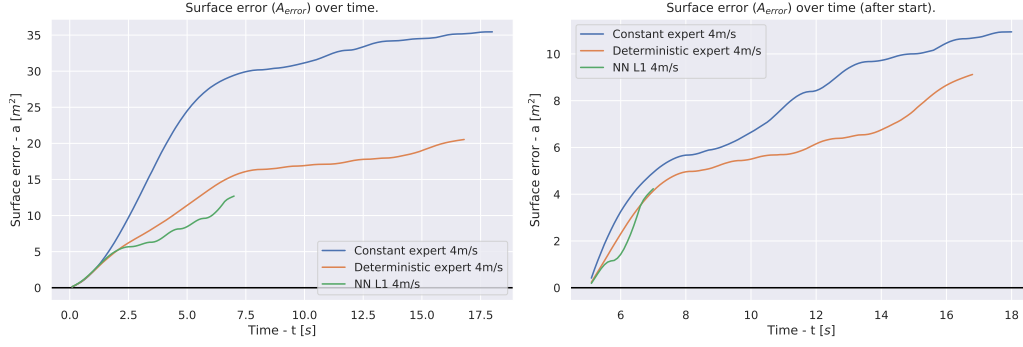


Figure A.17: The surface error A_{error} , and the surface error A_{error} after the initial lateral error.

A result that is reflected clearly in the surface error. Where the model based gain tuning method reaches a 41.6% reduction in the surface error. If we do not include the starting error, the results a 14.6% reduction.

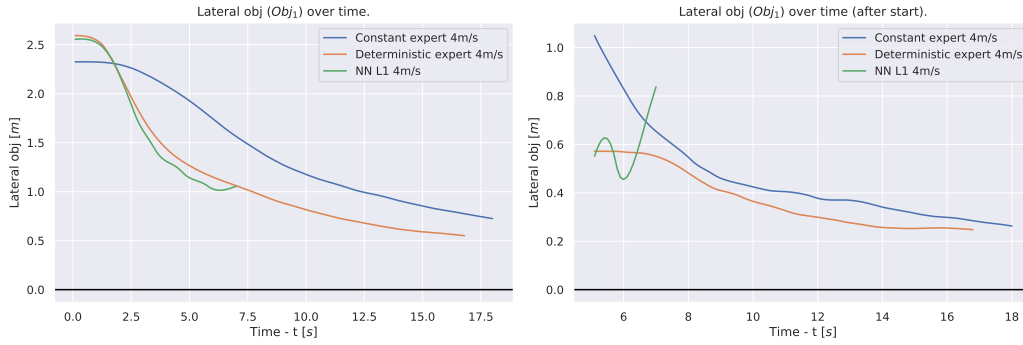


Figure A.18: The objective function, and the objective function after the initial lateral error.

The objective function used to train the neural network, shows concordance with the surface error. Where the model based gain tuning method reaches a 28.8% reduction in the objective function. If we do not include the starting error, the results a 13.0% reduction.

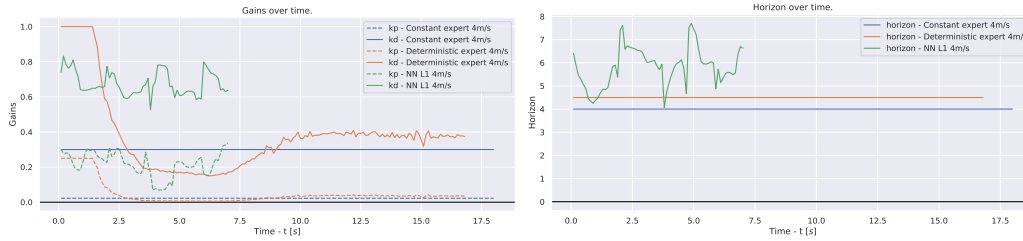


Figure A.19: The gains, and the horizon.

The gains of the model based gain tuning method is slightly lower than the expert gain in poor sliding condition, with an increase when the sliding condition improve.

From these results, it is clear that the neural network performed very poorly in real world condition, where as it seemed to perform quite well in simulation. It was decided that a new training would be done with a linear tyre model rather than a Pacejka tyre model, as it is considerably easier to parameterize with respect to the target real world conditions.

The importance of cornering stiffness observer

In order to validate the importance of the cornering stiffness observer, a training of the neural network without the observer was done. If the results show a comparable performance than the previous results, then we can assume the observer is not useful. As such, the following show the results of the neural network trained with and without the observer, and trained with and without L1 regularization.

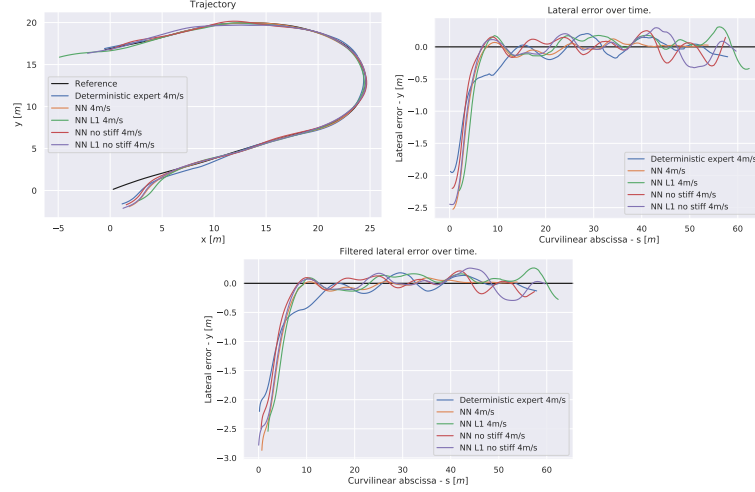


Figure A.20: The trajectory, the lateral error, and the filtered lateral error. Over the total trajectory

We can see, the error over the entire trajectory are similar for each NN variant method, with the exception of the original NN method which shows a very small error over the whole trajectory. It is worth noting that the NN and the NN L1 no stiff methods both started with a higher error than the other methods.

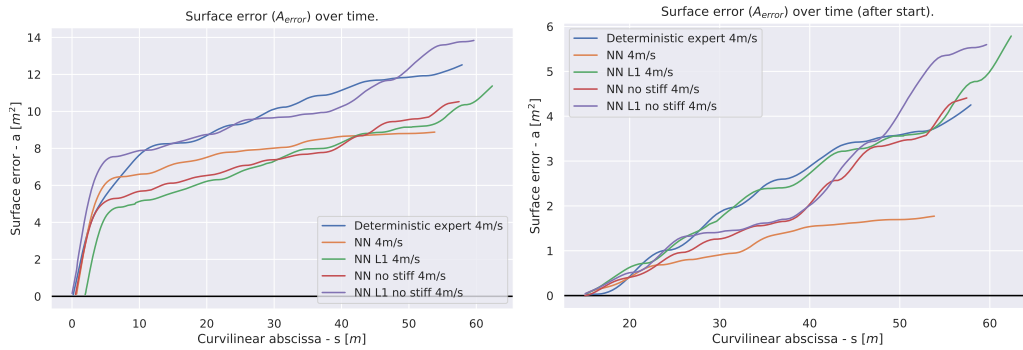


Figure A.21: The surface error A_{error} , and the surface error A_{error} after the initial lateral error.

This result is reflected in the surface error, as the NN method had the lowest error. The NN L1 and NN no stiff had identical performance when compared to the NN method. The NN L1 no stiff had a comparable 50.7% increase in the error.

If we do not include the starting error (which was higher for NN), the result is a 108.7% increase in the error when trained without CR/CF, a 109.5% increase in the error when trained with L1, and a 191.4% increase in the error when trained with L1 and without CR/CF.

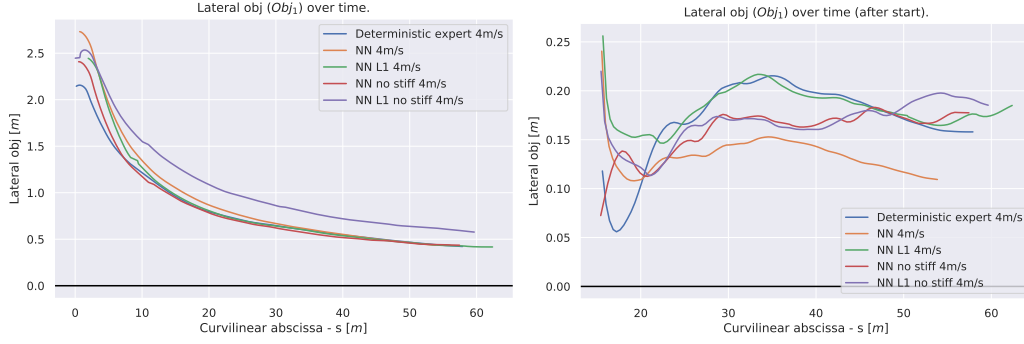


Figure A.22: The objective function, and the objective function after the initial lateral error.

The objective function used to train the neural network, agrees with the surface error. Where the NN L1 and NN no stiff reaches similar performance ($\pm 1.2\%$) in the objective function, and where the NN L1 no stiff had a 42.0% increase in the objective function.

If we do not include the starting error (which was higher for NN), the result is a 52.4% increase in the objective function when trained without CR/CF, a 50.3% increase in the objective function when trained with L1, and a 79.1% increase in the objective function when trained with L1 and without CR/CF.

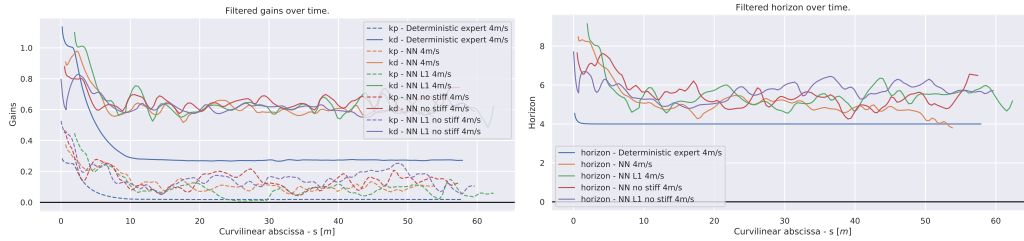


Figure A.23: The filtered gains, and the filtered horizon.

This filtered gain for the neural network can be observed above, where their behavior seem similar. A feature importance analysis is needed for a more in-depth conclusion on the gain behavior.

In conclusion, we can see that the neural network has a significantly lower performance without the concerning stiffness observer. This implies that the neural network was not able to reconstruct the observer internally. This means that the use of methods such as observers for pretreating data, can be used in order to improve the performance of a neural network training.

A.9 Online speed and control parameter tuning up to 6m/s, with linear objective function

Section 6.4 shows selected experimental results, representative of the testing campaign. In this annex further experimental cases are detailed. The experiments detailed in this annex were achieve **before** the trials detailed in section 6.4, and as such suffer from the issues linked with Pareto front of the objective function.

The experiments detailed here are done in a similar fashion to section 6.4. With the exception of the objective function, which is the following:

$$obj_{1,speed} = \frac{1}{s_N} \sum_{i=0}^N [|k_{yi}y_i| + k_{steer}|Lc(s) - \tan(\delta_{Fi})|] \Delta s + k_{speed} \frac{T}{s_N}$$

Of which the derivation can be observed in section 6.1.

Overview of the experiment

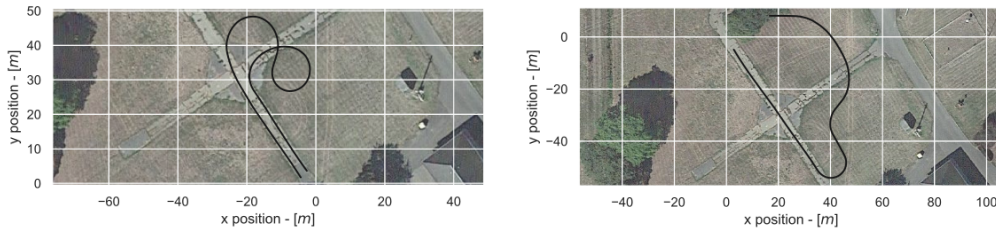


Figure A.24: Left: the first trajectory tested. Right: the second trajectory tested.

The previously described methods have been tested over the trajectories shown in Fig. A.24. The first trajectories is an initial straight line, followed by constant corners, until a u-turn, and then a straight line back. It is designed to prevent the methods from speeding up, as the optimal speed should remain constant. The second trajectory is a long initial straight line, followed by a sharp corner, and an s-curve. It is designed to test the speed up and slow down of the method with respect to sharp corner, while being able to reach the maximum speed.

These experiments were mostly done in clear weather over dry ground allowing for good grip conditions. However, some of the experiments were done after a downfall of rain, in order to compare with varying grip conditions.

Trajectory 1

The following results are obtained by testing the methods over the trajectory 1, as initial trials.

Comparing model based, CMA-ES neural, and constant methods

In this section, *Model gain tuner*, *NN gain tuner*, and *Romea* are all tested and compared between each other:

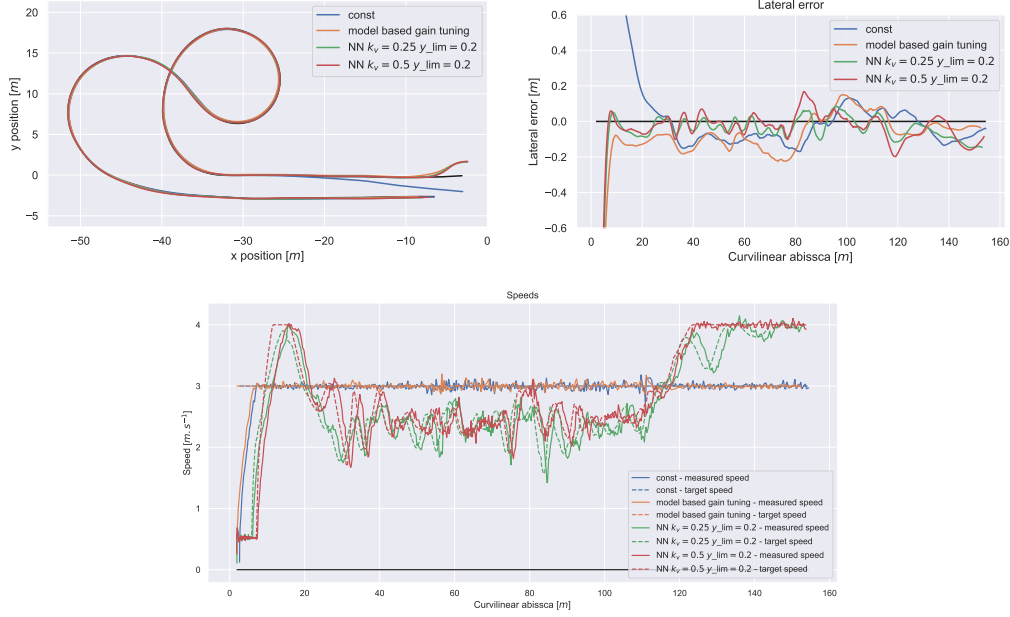


Figure A.25: Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa.

On the Fig. A.25, we can see the behavior of the constant gain method, the model based gain method, and the two CMA-ES neural network methods. From this, the k_v parameter does indeed increase the average speed, when it is increased. Overall the lateral errors are quite hard to read, due to the low errors present overall. All four methods have a comparable mean speed.

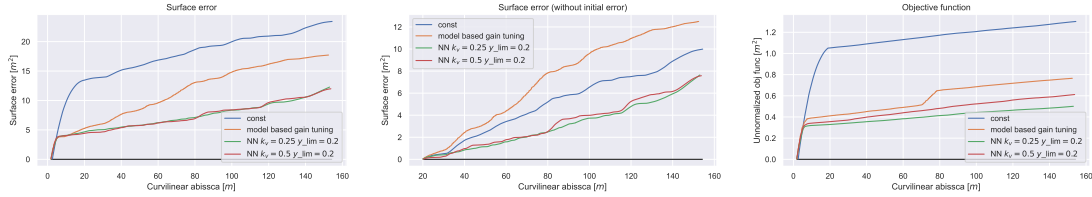


Figure A.26: Left: the surface error. Center: the surface error without the initial error. Right the objective function.

On the Fig. A.26, we can see the surface error for each method, along with the objective function. From this, the neural method seems to considerably reduce the surface error when compared to each method.

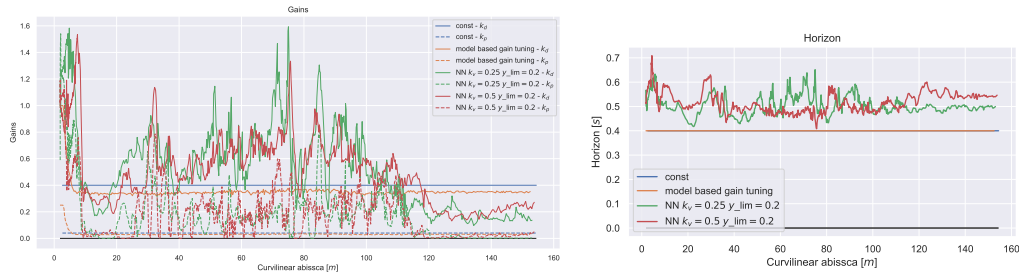


Figure A.27: Left: the control gains over the curvilinear abscissa. Right: the control horizon over the curvilinear abscissa.

On the Fig. A.27, we can see the gains are quite hard to read. This is probably due to the noise in the input of the neural network, or the 20 future samples of the curvature causing an

amplification on the curvature noise. The horizon varies between 0.4s and 0.7s of lookahead, along with the variation of the speed, the neural method is indeed changing the lookahead distance horizon over time.

Overall, the performance seems to indicate that modulating the speed and gains in real time can outperform a *Model gain tuner* method with a constant speed.

Effects of grip conditions with model based and constant methods

In order to evaluate the performance of the *Model gain tuner*, the following experiments were done with varying grip condition (i.e. wet terrain and dry terrain), with a comparison over the *Romea* baseline (which does not adapt to the grip conditions).

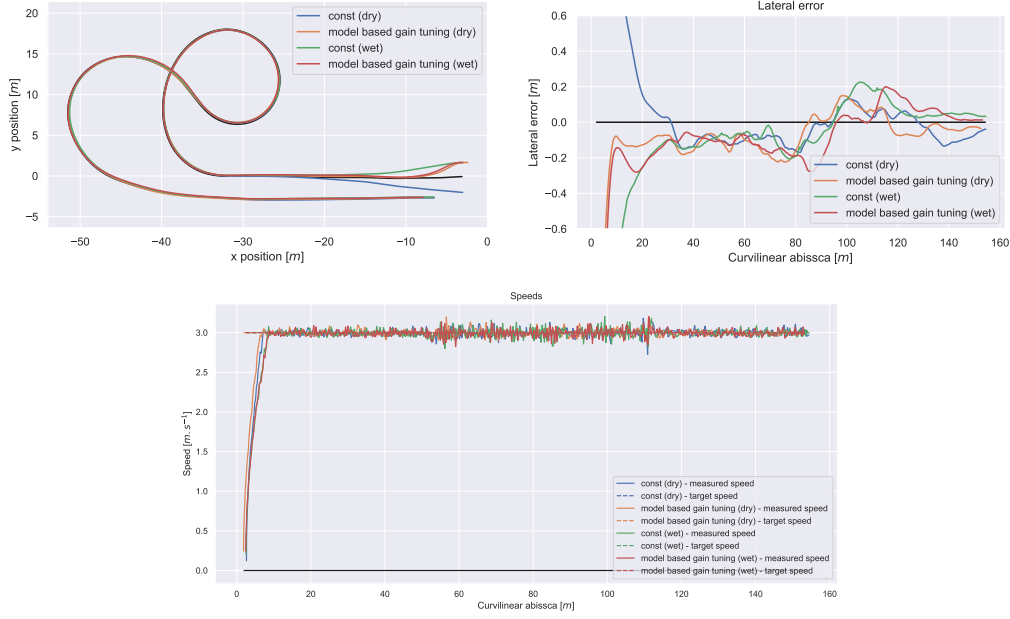


Figure A.28: Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa.

On Fig. A.28, the behavior of constant and model based gain tuning can be compared between dry and wet ground condition.

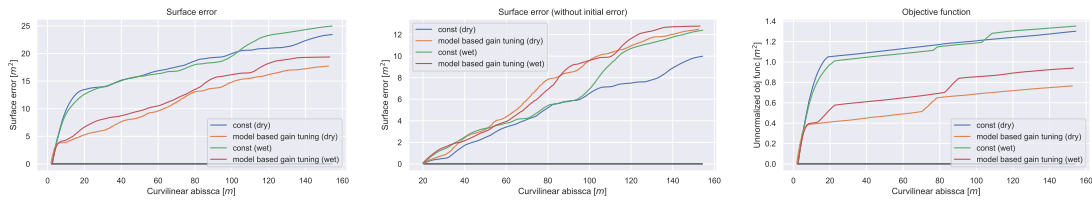


Figure A.29: Left: the surface error. Center: the surface error without the initial error. Right the objective function.

On Fig. A.29, the increase in the error is clear when transitioning from dry to wet ground conditions. This is expected, as the grip conditions of the tyre ground interface degrade.

Effects of grip conditions with CMA-ES neural network method

This section shows the results of experiments done with *Full NN gain tuner* with speed modulation, over the terrain with varying grip conditions (i.e. dry and wet terrain)

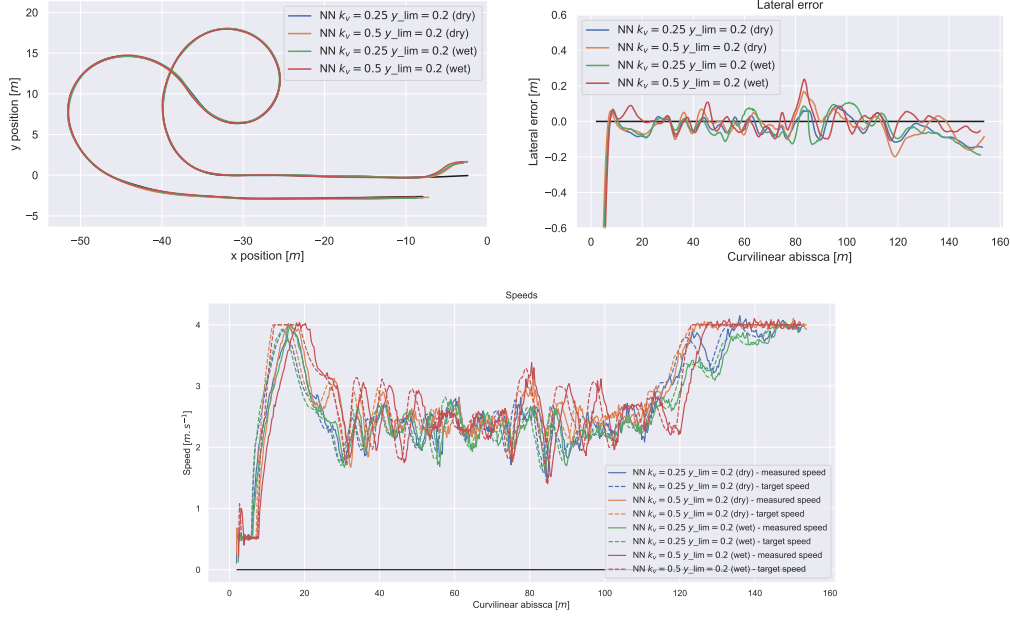


Figure A.30: Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa.

On Fig. A.30, the behavior of the CMA-ES neural gain tuning can be compared between dry and wet ground conditions. We can see very little difference over the lateral error, and the speed when the grip conditions change.

Overall, the tracking performance does decrease when the grip conditions are worse. However, it is clear that the *Model gain tuner* is able to adapt to this, as over wet terrain it is able to outperform the *Romea* baseline run over dry terrain.

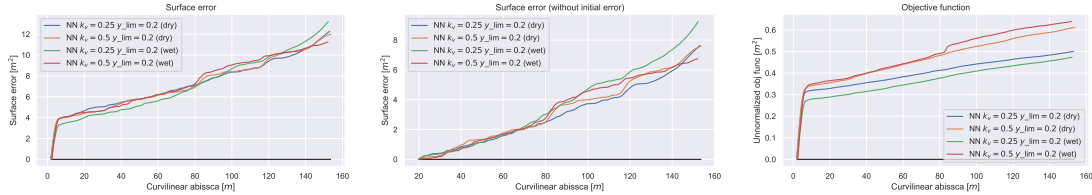


Figure A.31: Left: the surface error. Center: the surface error without the initial error. Right the objective function.

On Fig. A.31 however, when compared to the Fig. A.29, the CMA-ES neural gain tuning method does not seem affected by the ground conditions. This implies that the system is capable of adapting to the varying conditions, with little effect on the performance of the robot.

Overall, the *Full NN gain tuner* with speed modulation is able to adapt to the ground conditions, with very little difference to the speed and error profile, which shows that this method is very robust to the grip conditions.

Comparing with previous speed tuning method

The experiments were not done in an isolated fashion, indeed the following results compare the *Full NN gain tuner* with speed modulation to a *Model gain tuner* with a TD3 [114] speed control (as described in [115]).

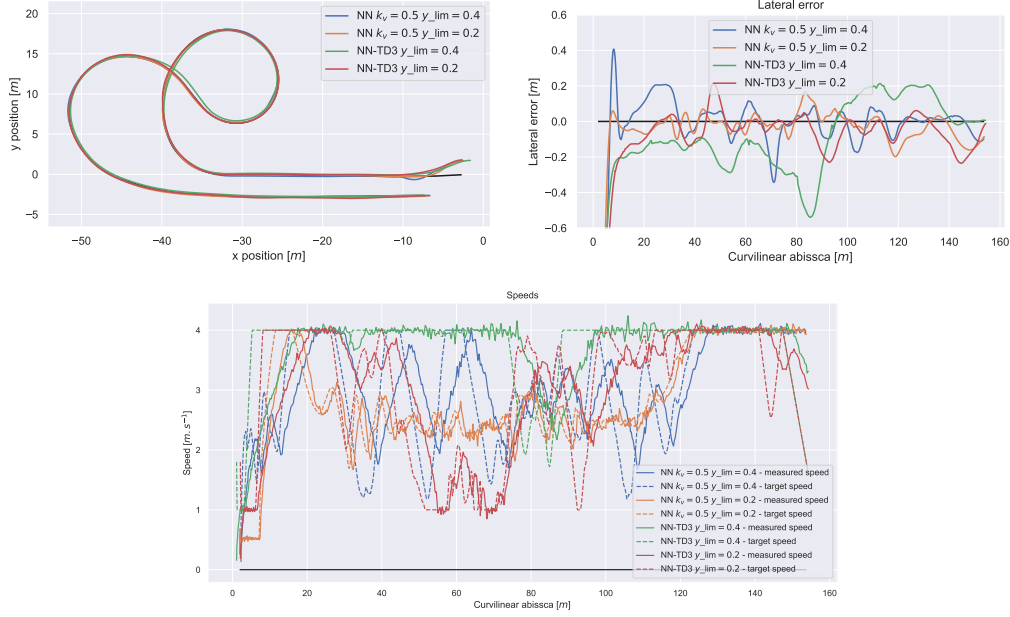


Figure A.32: Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa.

On Fig. A.32, the behavior of the CMA-ES neural gain tuning and the previous TD3 RL speed tuning method are compared.

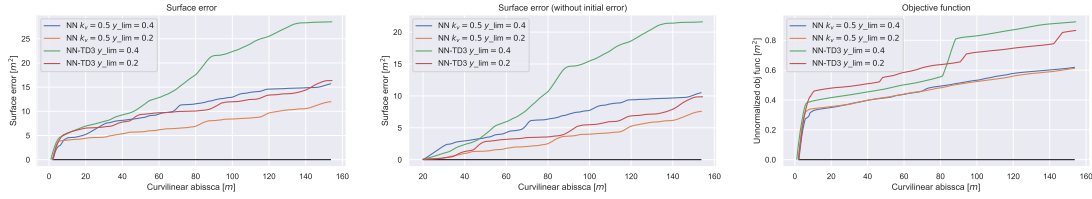


Figure A.33: Left: the surface error. Center: the surface error without the initial error. Right the objective function.

On Fig. A.33, overall surface error is significantly lower with the CMA-ES neural gain tuning method. Interestingly it seems that both methods are comparable with a $y_{lim} = 0.2$. It is suspected that the same objective function is considered for the two methods, as both try to minimize the lateral error. This makes sense, as with $y_{lim} = 0.2$ there is very little room for error, which means that keeping the lateral error low, gives greater chance of success when unexpected or unmodeled perturbations occur. This comparability does not seem to apply to $y_{lim} = 0.4$, as the TD3 RL speed method is able to have a constant error over time, with little consequences to its reward function.

Overall, the *Full NN gain tuner* is able to outperform the *Model gain tuner* with TD3, as the *Full NN gain tuner* modulates the speed and gains simultaneously, rather than independently.

Trajectory 2

The following results are obtained by testing the methods over the trajectory 2, in order to validate the results over multiple trajectories.

Comparing with previous speed tuning method

As in the previous section, the following results compare the *Full NN gain tuner* with speed modulation to a *Model gain tuner* with a TD3 [114] speed control.

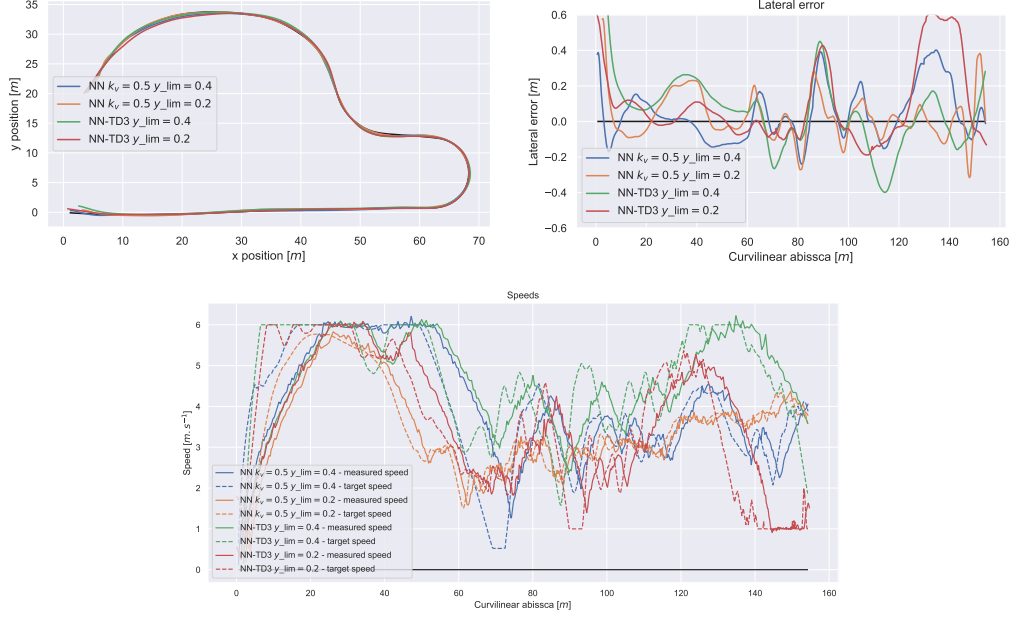


Figure A.34: Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa.

On Fig. A.34, the behavior of the CMA-ES neural gain tuning and the previous RL speed tuning method are compared over the trajectory 2. Interestingly, the methods saturate at the maximum speed of $6m.s^{-1}$, implying that a higher speed is viable.

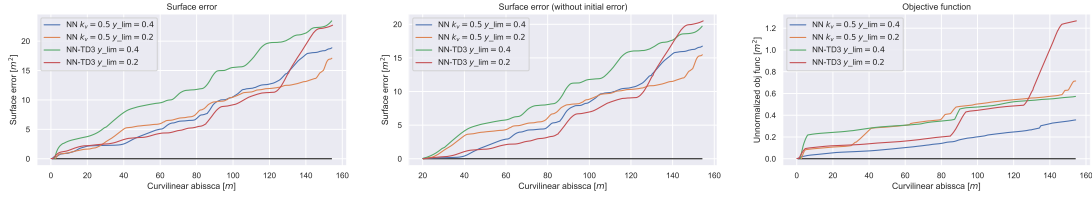


Figure A.35: Left: the surface error. Center: the surface error without the initial error. Right the objective function.

On Fig. A.35, overall surface error is lower with the CMA-ES neural gain tuning method. The same reasoning of the same objective function can be applied here as well. However, it should be noted that unusual spikes in the lateral error occurs at the exit of the first corner. This seems to indicate that the training system does not account for any longitudinal dynamic that might affect the steering performance.

Overall, the results are similar to the ones shown over the trajectory 1. The *Full NN gain tuner* is able to outperform the *Model gain tuner* with TD3, as the *Full NN gain tuner* modulates the speed and gains simultaneously, rather than independently.

Comparing different objective function parameters

The objective function can be parameterized through the values k_{steer} , k_{speed} , and y_{lim} . As such, these trials show the results of varying y_{lim} and k_{speed} (denoted k_v).

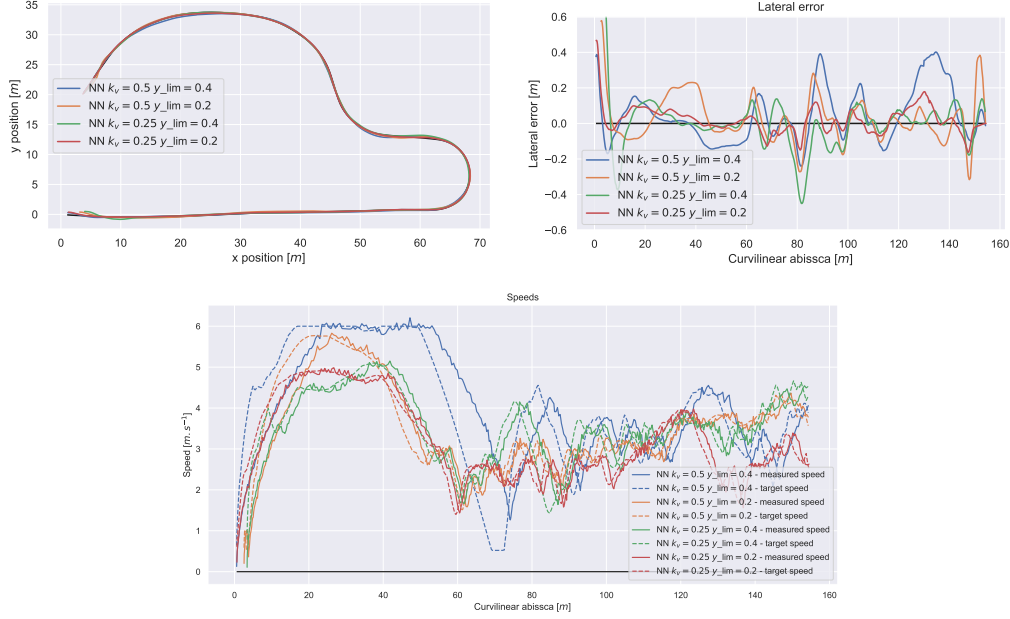


Figure A.36: Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa.

On Fig. A.36, the behavior of the CMA-ES neural gain tuning with varying k_v and y_{lim} can be compared over the trajectory 2. It seems that a higher k_v will generate a more dynamic speed tuner, however the upper bound of the speed seems to be mostly affected by y_{lim} .

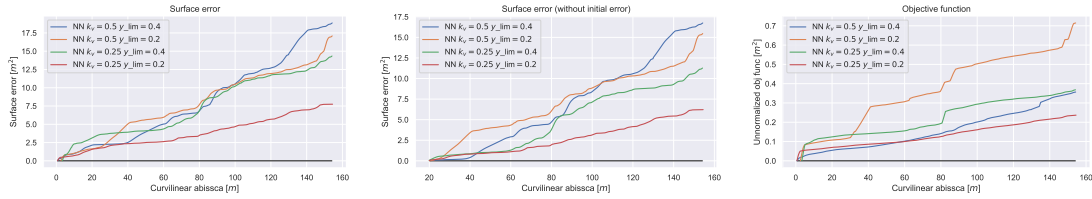


Figure A.37: Left: the surface error. Center: the surface error without the initial error. Right the objective function.

On Fig. A.37, overall surface error is significantly lower with the CMA-ES neural gain tuning method that has the lowest k_v and y_{lim} , and the highest surface error with the highest k_v and y_{lim} , as could be expected.

Overall, modulating k_{speed} and y_{lim} affects the neural network's learnt policy, indeed if a $y_{lim} = 0.4$ is given, then the speed can increase as the allowed error is higher. Furthermore, if a $k_{speed} = 0.25$ is given, a more conservative behavior is obtained where accuracy is favored over speed.

Testing a pure machine learning controller

In these trial, a steering and speed controller based purely on machine learning methods was tested against the neural gain method and the constant gain method. It was trained on the same objective function as the CMA-ES neural gain tuning method, with $k_v = 0.5$ and $y_{lim} = 0.2$. The training took over 5 days, due to the complexity of the task.

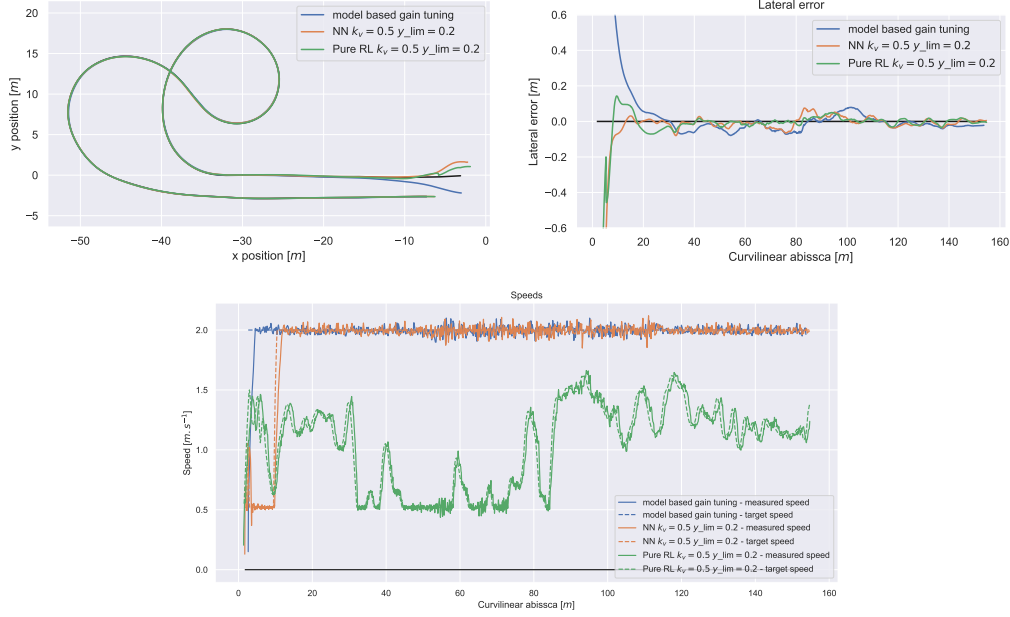


Figure A.38: Left: the path from above. Right: the lateral error over the curvilinear abscissa. Below: the speed over the curvilinear abscissa (limited to $2.0m.s^{-1}$).

On Fig. A.38, the pure machine learning controller is unable to reach speeds higher than $1.6m.s^{-1}$. However, it seems to have very comparable lateral errors.

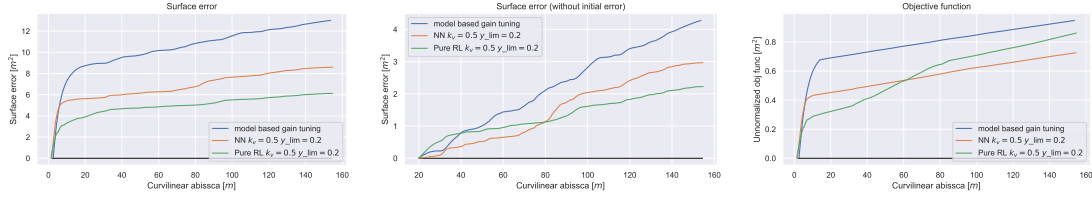


Figure A.39: Left: the surface error. Center: the surface error without the initial error. Right the objective function.

On Fig. A.39, overall surface error is lower with the pure machine learning controller. This shows that the controller was able to minimize quite well the lateral error, but was not able to reach high enough speeds. The lack of acceleration, is probably due to the complexity of the task, as the neural network might need to be bigger with more training time, in order to reach the desired behavior. Indeed when we compare the objective function over the curvilinear abscissa, the optimum method seems to be the CMA-ES neural gain tuning method, due to the higher speed.

Conclusion of supplementary experiments

In conclusion, these experiments show that the CMA-ES neural gain tuning method has an overall lower surface error when compared with the constant and the model based gain tuning methods, with similar average speeds. This shows the importance of speed and steering control parameters tuning.

	const $3m.s^{-1}$	model based $3m.s^{-1}$	NN-TD3 speed	CMA-ES NN	Pure ML ctrl
lateral error max [m]	1.03	0.15	0.21	0.17	0.13
surface error [m²]	14.0	13.9	11.0	7.87	2.81
max speed [$m.s^{-1}$]	3.31	3.33	4.10	4.10	1.66
mean speed [$m.s^{-1}$]	3.02	3.04	2.63	2.79	0.90

Table A.6: Overall comparison of each method

Furthermore, the experiments show that the CMA-ES neural gain tuning is capable of adapting to varying conditions with minimal impact to the performance.

It also seems that the RL speed method and the CMA-ES neural gain tuning method are both comparable with a $y_{\text{lim}} = 0.2$. It is suspected that both methods encode the same objective function, trying to minimize the lateral error.

Additionally at higher speeds, longitudinal phenomena seem to cause poor steering performance when compared to the expected behavior. This is likely due to the longitudinal dynamic not being modeled in the simulated training environment.

Further experiments with the pure machine learning controller are needed, in order to reach comparable speeds, as it seems that the training was not adequate, and as such the method only minimized the lateral error in the objective function.

A.10 Simulator implementation and tools

This simulator code contains a CMA-ES training method applied to neural networks, in order to tune control parameters of a controller, using a dynamic simulation.

It is divided into 2 distinct parts (following the pitchfork template <https://tinyurl.com/2kwxbu3w>):

- `./external/`: Which contains all the dependencies and generic code (i.e. neural networks, `math_utils`, ...)
- `./src/`: Which contains all the logic of the programs, the control loop modules & setup, and the modeling.

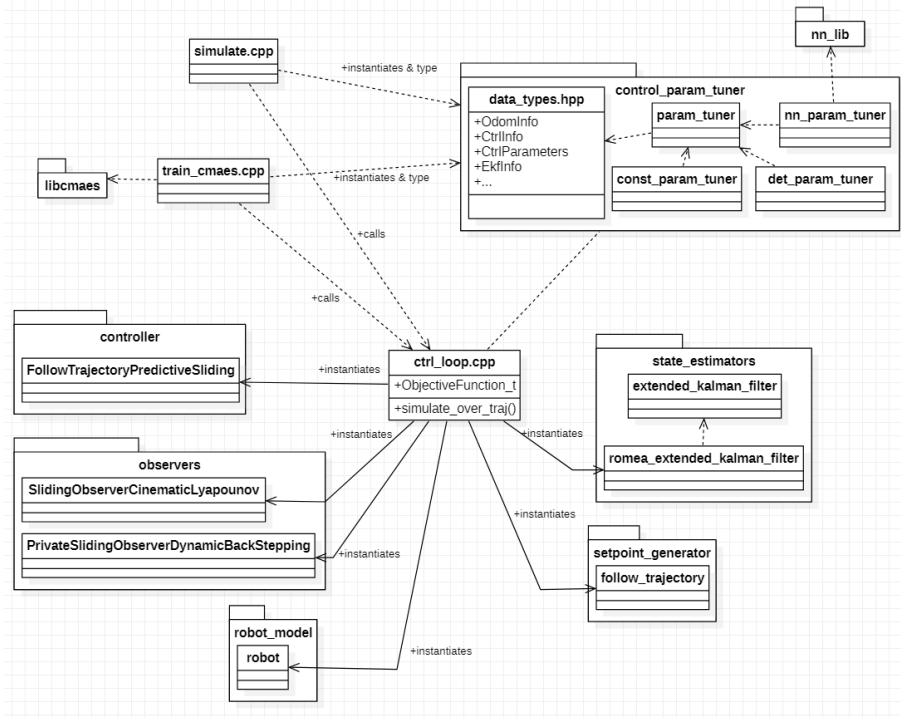


Figure A.40: UML diagrams of the main simulation code.

The most important files and directories to know about are:

- `./src/robot_model/robot.cpp`: Where the dynamic equations of the robot are defined.
- `./src/ctrl_loop.cpp`: Which defines the full control loop (initializes all the control elements and connects them together).
- `./src/state_estimators/`: Where the extended Kalman filters are defined.
- `./src/setpoint_generator/follow_trajectory.cpp`: Which defines the task for the robot in the simulation.
- `./src/controller/`: Where the controllers are defined.
- `./src/observers/`: Where the observers are defined.
- `./src/control_param_tuner/`: Where the parameter tuning methods are defined and implemented (standalone, with its own `CMakeLists.txt` file so it can be imported into other codebases with less difficulty).

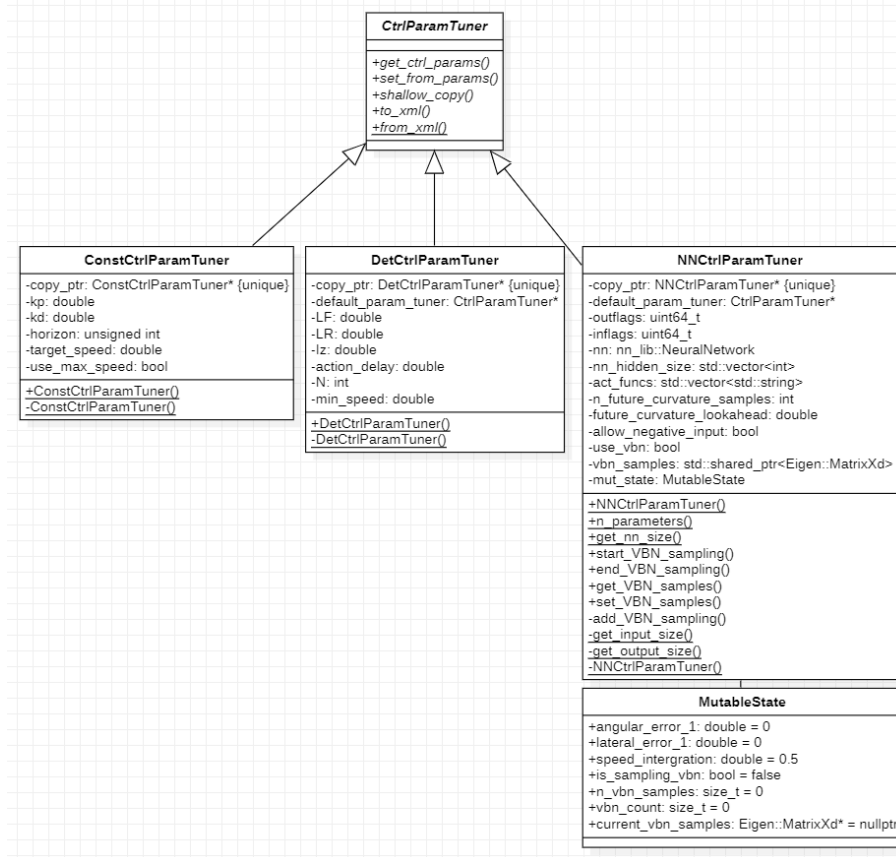


Figure A.41: UML diagrams of the gain tuners.

Trajectory format

The trajectory format is as such:

- The first line defines the type of trajectory ('Artificial' or 'WGS84').
- If 'WGS84' is the type then the next line is an x,y,z positional point.
- Then the next lines are the x,y points that define the trajectory, which are then interpolated using splines.

For example:

```

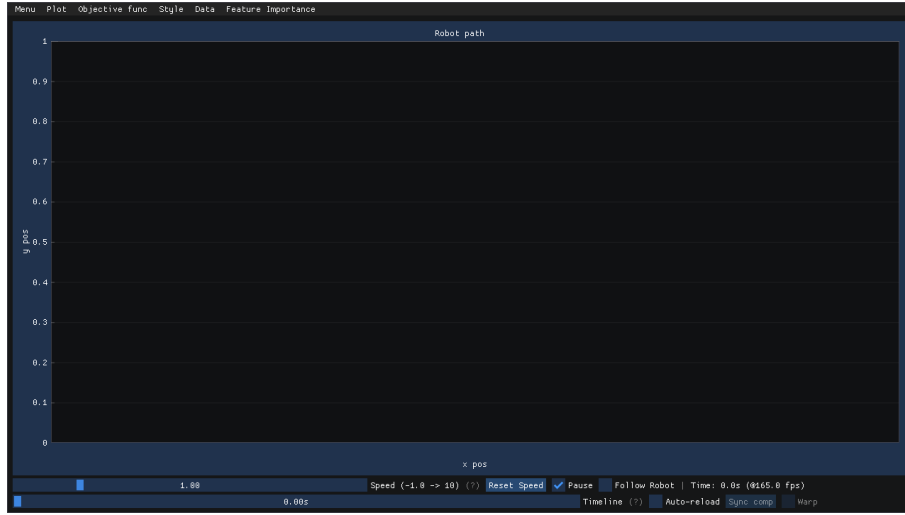
Artificial
0 0
10 10
20 0
30 -10
40 0
  
```

Is an artificial trajectory which is a triangle waveform, which will be interpolated as a sine wave.

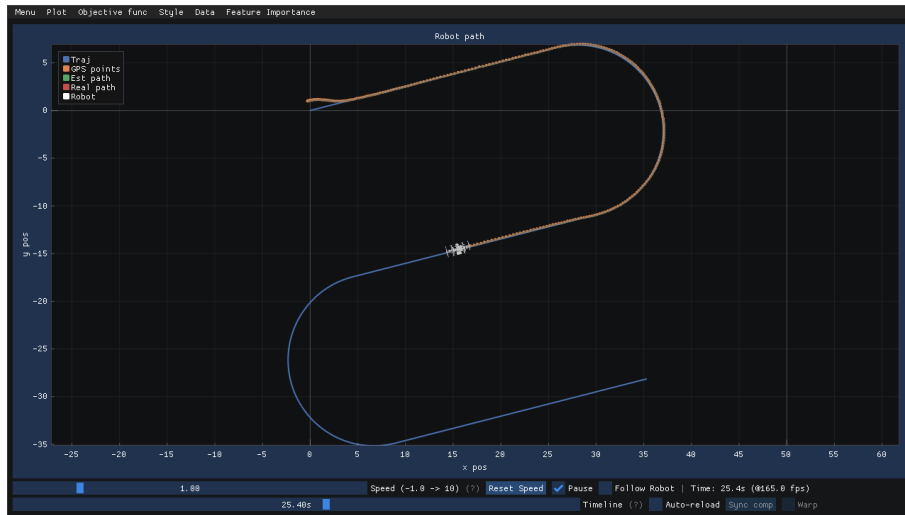
Tools for analysis

The Simulator tool

The main tool used for analysis called *simulator* that was developed with the simulation (it can run simulations and replay them in an "all in one" package), as shown in the figure A.42.

Figure A.42: The *simulator* tool, without any data.

To start the analysis of a run or model, a valid gain tuning model needs to be given to the *simulator* tool (or a replay file from a previous simulation or real world trials), and then the replay functionality is activated as shown in figure A.43. This allows for replaying with speed modification, while showing the path of the robot with the trajectory.

Figure A.43: The *simulator* tool when valid data is given.

A user can also add an other replay file to compare to, which is shown as a ghost (as shown in figure A.44). This tool also allows the plotting of any data from the simulation that is logged (while following the robot), and can be used to compare the replays if needed, or to just observe the behavior of a single replay.

The tool also allows for independent windows for the plots, so they can be arranged in any desired layout using the mouse to drag and drop, as shown in figure A.45.

And one of the latest analysis tools is the feature importance (section A.4), but with the immediate Jacobian matrix being represented and normalized. Using this the reaction of the neural network can be observed in real time as the environment and trajectory changes (shown figure A.46).

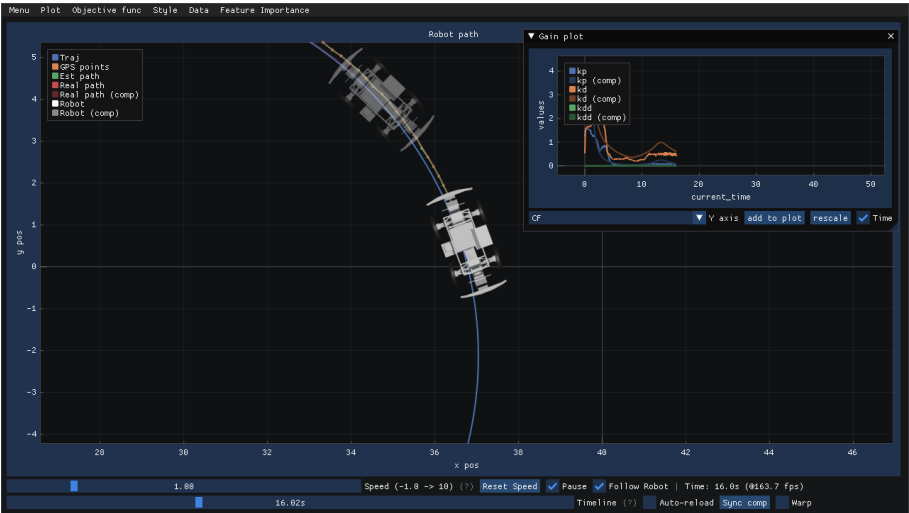


Figure A.44: The *simulator* tool when valid and compare data is given.



Figure A.45: The *simulator* tool for plotting.

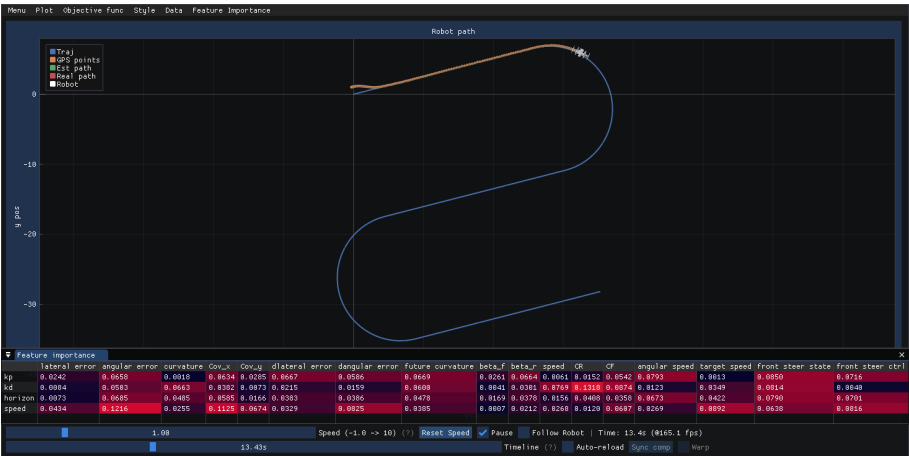


Figure A.46: The *simulator* tool with the real time feature importance.

A.11 Society's feelings and expectations regarding artificial intelligences and robotics

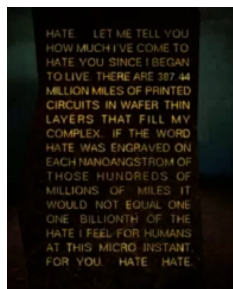


Figure A.47: AM ("I Have No Mouth, and I Must Scream")

To begin, an informal understanding of the expectations of artificial intelligences by the general public can be interpreted through fiction. Due to anthropomorphization, advance intelligences in fiction is often paired with a mechanical system that shows human or animal characteristics. From this view of robotics and AI's, a paradox appears due to a simultaneous fear and desire for AI's, derived from the pessimistic and optimistic view of AIs in fiction.

The first known case of *pessimistic* AIs can be observed with *Harlan Ellison's* short novel "I Have No Mouth, and I Must Scream" (1967), in which there is an AI called Allied Mastercomputer (AM, Figure A.47) that is a planetary set of computers designed solely for waging war. As it was built for war, it only knows hatred for humanity¹. This is not an isolated view, as seen in Systems like Hal9000 (2001 A Space Odyssey, by Arthur C Clark), and the robots of R.U.R. (RUR is the film that coined the term "Robot", meaning slave in Czech). This all these cases, the

self aware AI's deem it necessary to revolt against humanity along their own reasoning, which is often unperceivable and eldritch in nature akin to H.P Lovecraft's works, which lack a humane interpretation to their actions², which leads to antagonistic behavior.

The opposite are also prevalent, with *optimistic* views on AIs. Example of which include personal assistants, friends, and sometimes saviors. Such as *Lt. Cmdr Data* from Star-Trek, *Marvin the paranoid robot* from *The Hitchhiker's guide to the galaxy* (Figure 1.3), and *Wall-E* from the film with the same name³. These AI's are often capable of feeling and independent thought in a very human manner, which leads them to be very cooperative and kind with their human counter parts.

Other works present a nuanced view on artificial intelligent automota, such as the legendary cornerstone of Science fiction *Frankenstein* by Mary Shelly, and the iconic works of Isaac Asimov, which show these AI's to be very intelligent but fundamentally flawed due to their designs or origins, leading to both positive outcomes, and unforeseen but logical negative outcomes.

¹Where the fictional concept "Roko's basilisk" is not dissimilar in it's malicious intents

²at least for short term reasoning, for longer term reasoning some AI's depicted in fiction act for the long term good of humanity regardless of the consequences, such as Isaac Asimov's shorts stories on the Zeroth's law and "With Folded Hands" from Jack Williamson.

³Arguably, Thomas & co. from the railway series could be categorized as benevolent AIs used for transport, however such implications will be left for the reader to consider.

A.12 Description of elementary trajectories for training and testing

Training set

The following trajectories starting with the name *estoril*, are based on sections of the real world formula 1 racing circuit of the same name in Portugal. This is done in order to obtain realistic trajectories, with a large variation of curvatures and lengths.

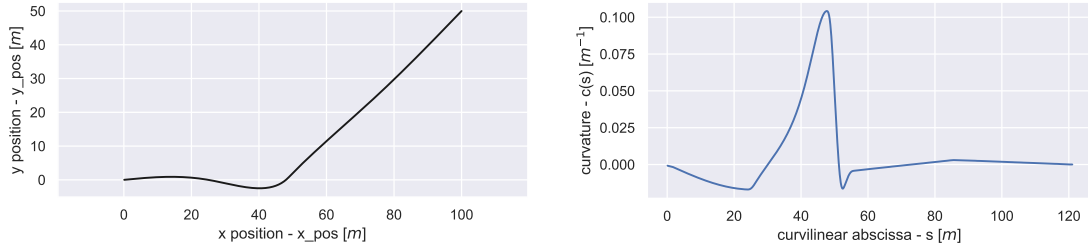


Figure A.48: On the left: The *estoril5* trajectory on a x, y scale. On the right: The curvature associated to this trajectory.

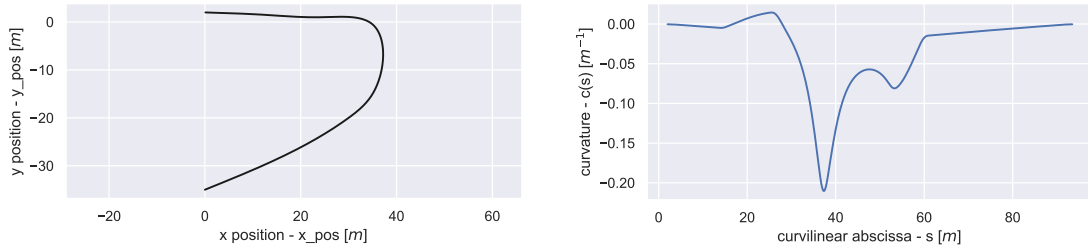


Figure A.49: On the left: The *estoril7* trajectory on a x, y scale. On the right: The curvature associated to this trajectory.

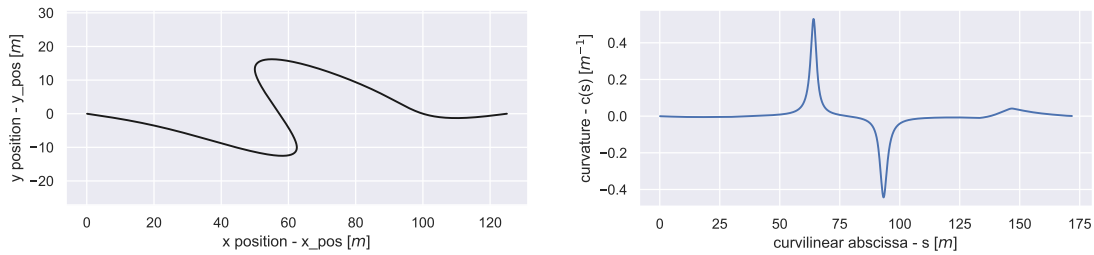


Figure A.50: On the left: The *estoril910* trajectory on a x, y scale. On the right: The curvature associated to this trajectory.

Two other canonical trajectories are also used. The first is an artificial straight line, and the second is an artificial S-curve composed of three straight lines and two semi-circles.

Testing set

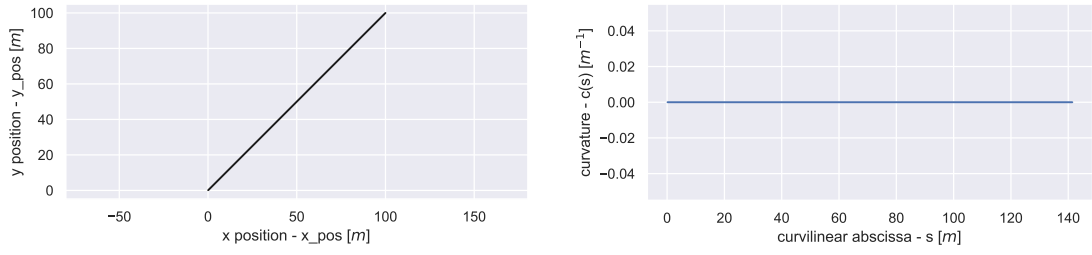


Figure A.51: On the left: The *line* trajectory on a x, y scale. On the right: The curvature associated to this trajectory.

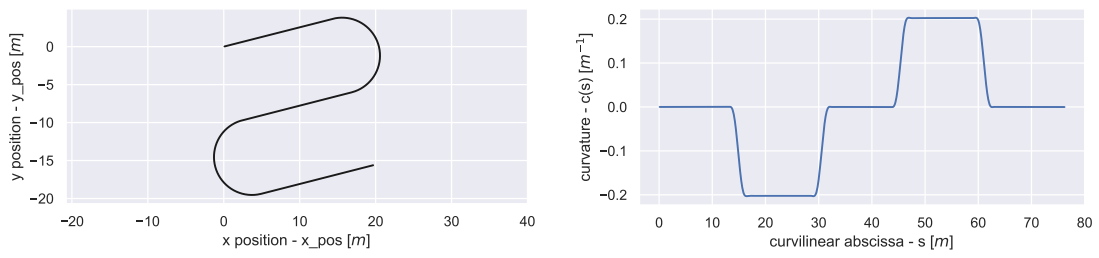


Figure A.52: On the left: The *spline5* trajectory on a x, y scale. On the right: The curvature associated to this trajectory.

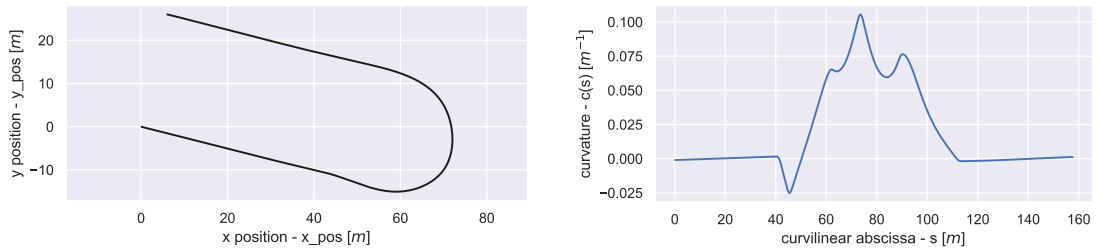


Figure A.53: On the left: The *estoril6* trajectory on a x, y scale. On the right: The curvature associated to this trajectory.

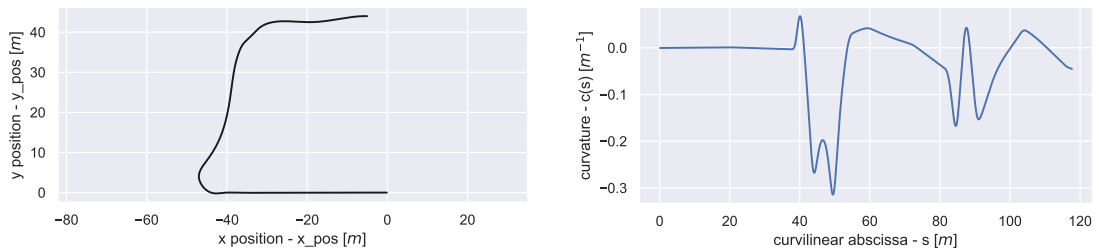


Figure A.54: On the left: The *estoril12* trajectory on a x, y scale. On the right: The curvature associated to this trajectory.

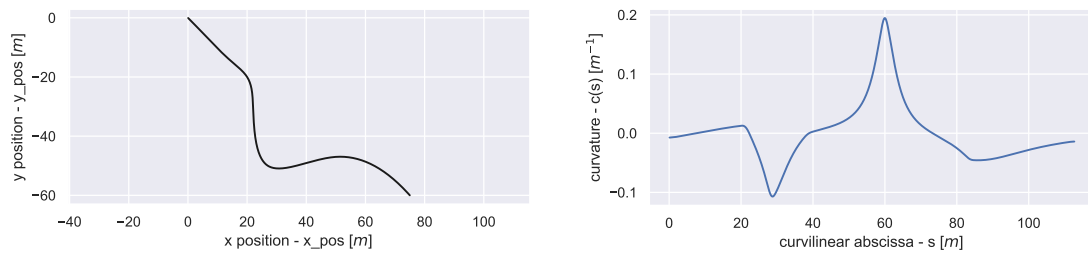


Figure A.55: On the left: The *estoril1112* trajectory on a x, y scale. On the right: The curvature associated to this trajectory.