POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING

Master degree course in Mechatronic Engineering

Master Degree Thesis

Model Predictive Control with a Learned Model for Safe Exploration

Supervisor

Prof. Alessandro Rizzo

Candidate

Alessio Rosa

ID Number:244010

Company Supervisor

Dott. Andrea Lonza

Academic Year 2021/2022

Table of contents

List of Figures

Nomenclature

Main Acronyms	
MPC	Model Predictive Control
DNNs	Deep Neural Networks
ANNs	Artificial Neural Networks
AI	Artificial Intelligence
ROS	Robotics Operating System
UGV	Unmanned Ground Vehicle
CMDPs	Constrained Markov Decision Processes
CEM	Cross-Entropy Method
RM	Random Shooting
MF	Model-Free
MB	Model-Based
MSE	Mean Squared Error
RMSE	Root Mean Squared Error
MAE	Mean Absolute Error
ReLu	Rectified Linear Unit Function
GPUs	Graphics Processing Units
OOP	Object-Oriented Programming
ADAM	Adaptive Moment Estimation
RMS	RMSprop
SGD	Stochastic Gradient Descent
RL	Reinforcement Learning

Abstract

Modern information technologies and the advent of machines powered by Artificial Intelligence (AI) have already strongly influenced the world of work in the 21st century. Advances in Artificial Intelligence technology and related fields have opened up new markets and new opportunities for progress in critical areas such as health, education, energy, economic inclusion, social welfare, and the environment. In recent years, machines have surpassed humans in the performance of certain tasks related to intelligence [1]. Although nowadays it is hard to think that AI can outperform human intelligence in a broadly-applicable field, it has been demonstrated that machines have reached and exceeded human performance in many narrowly defined and structured tasks (AlphaGo Zero, a Deep Mind's algorithm, beat human capabilities becoming the strongest Go player).

Through the use of artificial intelligence, robots will be able to independently assess what is happening around them and make decisions on the actions they need to take. For this reason, the ability to *robustly* guarantee safety becomes absolutely essential: for instance, consider tasks such as steering an autonomous vehicle along a given reference track, precise machine tooling, or control of any autonomous system that interacts with humans. All these applications have in common that the autonomous system should be able to complete the task without harming itself and its surroundings since in real contexts safety is a critical factor and errors are not acceptable.

This Thesis studies the Safe Exploration Problem without assumption about prior knowledge of the system dynamics and the constraint function. To cope with this problem we combine a medium-size Deep Neural Network model with Model Predictive Control to achieve excellent sample complexity and sample efficiency in order to produce plausible gaits to accomplish locomotion tasks. The Model Predictive Control uses the random shooting methods to optimize the control sequence considering the dynamics of the model and the constraints learned by the Deep Neural Networks. We evaluate the performance of the Robot in Safety Gym

environment that provides a set of tools for safe exploration research.

The results show that our approach achieves better constraint satisfaction and also a better sample efficiency when compared to model-free and unconstrained algorithms.

Introduction

Modern information technologies and the advent of machines powered by Artificial Intelligence (AI) have already strongly influenced the world of work in the 21st century. Advances in Artificial Intelligence technology and related fields have opened up new markets and new opportunities for progress in critical areas such as health, education, energy, economic inclusion, social welfare, and the environment [1]. In recent years, machines have surpassed humans in the performance of certain tasks related to intelligence.

Artificial Intelligence is "embodied" in robots, which are complex systems integrating many AI technologies such as vision, natural language, study of the movement, communication, machine learning, and knowledge representation and planning. With artificial intelligence, robots will be able to independently assess what is happening around them and make decisions on the actions they need to take. The development of artificial intelligence and machine learning technologies and their application in robotics is a prerequisite for the creation of really useful and smart robots.

The need to handle complexity is a crucial point in modern control design, especially in robotics for different reasons: firstly, due to complex tasks or system descriptions that are high-dimensional and nonlinear; secondly, due to stability requirements or the satisfaction of hard constraints on inputs and states. In particular, the need to *robustly* guarantee safety becomes absolutely important in the case of human interaction or people involved within the process, such as for automated driving or human-robot interaction [2]. Indeed, the new trends in robotics research is getting robots closer to human social needs. With the increase of social interaction between humans and robots in everyday and working life, it is necessary to pay great attention to the problem of safety of robots and its surroundings. For safe autonomous functioning in a dynamic unstructured environment, a robot should possess a capability of real-time data processing under information uncertainty. To this purpose, an advanced artificial intelligence control software is needed to allow

robots working with undirected objects of arbitrary shape, to interact with the external environment, to perform the required sequence of operations in a changing environment, avoiding risks for humans.

To this purpose, an approach consists of combining novel robust model predictive control (MPC) with function approximation via (deep) neural networks. Specifically, the aim of the thesis is to propose a new robust setpoint tracking MPC algorithm, which achieves reliable and safe tracking of a dynamic setpoint while guaranteeing stability and constraint satisfaction. The proposed controller is able to cope with the local optimization, the constraint control, and the direct control for a given (possibly changing) target, verifying the constraints while guaranteeing the robust stability of the system. Model predictive controllers are capable of regulating the controlled variable ensuring constraint satisfaction. This work of thesis is focused on two main steps. Firstly, a constrained MPC model-based algorithm is presented, which achieves near-optimal task performance, ensuring a low constraint violation rate. The problem is formulated under the constrained Markov Decision Process framework, with no assumption regarding the system dynamics and constraint function that are learned from collected data. Secondly, the proposed method alternates between running the MPC to attempt the task and collect data at training time, under partially state observations provided by a random training environment, and using this data to train a deep neural network that is used to model the system dynamics.

Therefore, in the first chapter a safe learning-based model predictive control scheme for nonlinear systems with state dependent uncertainty is covered along with a constrained Markov decision process framework. It is an effective method for controlling robotic systems because of its robustness to moderate model errors, ability to use high-level objectives, and relative simplicity.

The method guarantees the existence of feasible return trajectories to a safe region of the state space at every time step with high probability. Specifically, the system uncertainty is estimated based on observations through statistical modelling

techniques, such as the random shooting one. The algorithm can safely learn about the system dynamics when it is combined with an exploration scheme. Thus, the concept of safe exploration is also covered, with the aim of reducing the uncertainty about the model.

The second chapter, instead, covers some of the Model-based techniques, the approach to dynamic model learning using Artificial Neural Network, some of the possible challenges that may be encountered when modeling the environment, and the pros and cons of using a Model-based approach.

In the third chapter, instead, an overview about the simulation environment and the main framework used for the neural network design is addressed. It also provides an analysis of the simulation results carried out in the Safety Gym environment, where the robot navigates to a certain goal while avoiding all of the hazard areas according to safety requirements and task performance satisfaction.

Lastly, the fourth chapter summarizes the conclusions and final considerations.

CHAPTER 1

Optimal control is a subject where it is desired to define the inputs to a dynamical system for optimizing a specified performance index while satisfying any constraints on the motion of the system [3]. The goal of control is to make a dynamical system behave in a certain way or to solve a specific task and guarantee the satisfaction of safety constraints. Safe deep learning problems are usually modelled under the constrained Markov Decision Processes (CMDPs). Thus, a formal introduction of the MDP optimization problem is presented in this chapter, with a focus on learning the dynamics model from data. To this purpose, a model-based approach is considered, which is based on the Model Predictive Control technique, without assumptions about prior knowledge of the system dynamics and the constraint function [4]. Model Predictive Control (MPC) is a closed-loop control technique which uses a cost function and a learned or predefined model of the system which can be updated based on errors. It has been proven that the maximum MPC performance error is bounded and proportional to the model errors. Hence, the use of MPC is ideal when dealing with complex dynamics. Models used in MPC can be defined manually or learnt by collecting data from the system. With respect to model-free methods, which directly learns a value function (an estimation of how good it is for the robot to be in a certain state or how good it is to perform a certain action in that state), by interacting with the environment, the model-based ones use interactions with the environment to learn a model of it (the difference between the two main methods is covered in detail in Chapter 2). Indeed, since the environment dynamics and the analytical expression of the constraint function are unknown, both the dynamics model and constraint model have been defined by collecting training data during environment exploration executing random agent actions [15]. However, implementing MPC on fast dynamical systems with limited computation capacity remains generally challenging, since MPC requires the solution of an optimization problem at each sampling step. An approach to reduce computation load of MPC is by means of function approximation, where the control law is encoded, for example, through a Deep Neural Network (DNN) [5]. Thus, the learned environment

dynamics are parametrized through deep neural networks and the resulting models can then be used for model-based control, performed using MPC with a simple random-sampling shooting method. In shooting algorithms, the agent randomly generates action sequences, uses the dynamics to predict the future states, and chooses the first action from the sequence with the best expected reward [6]. The last section of the chapter, instead, summarised many recent approaches on how to define *safety* in the framework of optimal control. A novelty definition of safety is proposed, which divides the state space to *safe, critical* and *unsafe* states.

1.1 Constrained Markov Decision Processes

Markov decision processes (MDP) are discrete-time stochastic control processes used for modelling decision-making problems where the outcomes are partly random and partly controllable. They are useful for studying optimization problems in many disciplines like robotics and automatic control. Markov decision processes can be defined as

$$\{S, A, f, r, p(s_0), \gamma\} \qquad (1)$$

Where S represents the state space (the state is the current situation of the robot), A defines the action space (the action is the choice that the robot makes at the current time step t), f is a *transition function* such that $f: S \times A \to p(S)$, $p(S)$ defines the state-transition probability, and r is the *reward function* $r: S \to \mathbb{R}$ and $\gamma \epsilon$ [0,1] represents a discount parameter. At each timestep t (discrete global clock) some states $s_t \epsilon S$ are observed and an action $a_t \epsilon A$ is picked. Then, the environment returns a next state $s_{t+1} \sim f(\cdot | s_t, a_t)$ and associated scalar reward $r_t = r(s_t, a_t, s_{t+1})$. The first state s_0 is sampled from the initial state probability distribution $p(s_0)$. Within the environment, the robot acts according to a *policy* $\pi: S \to p(A)$, also known as a *contingency plan* or *strategy* [7]. In other words, a policy represents the learning robot's way in pursuit of goals at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states [8]. By repeatedly selecting actions and

transitioning to the next state, it is possible to sample to *trace* through the environment. To summarise, the robot controls the dynamic system at discrete decision steps. At each decision step, it observes the state of the system and determines the next action to take based on the observation and a policy. A policy maps states to actions and it is the robot's task to determine a policy that makes the system behave in an optimal way.

Figure 1 represents a block diagram of the main elements involved in a Markov Decision Process. A node represents a state. An arc represents a transition from one state to another under a certain action. An arc is labelled with a transition probability and a reward obtainable under the transition [9].

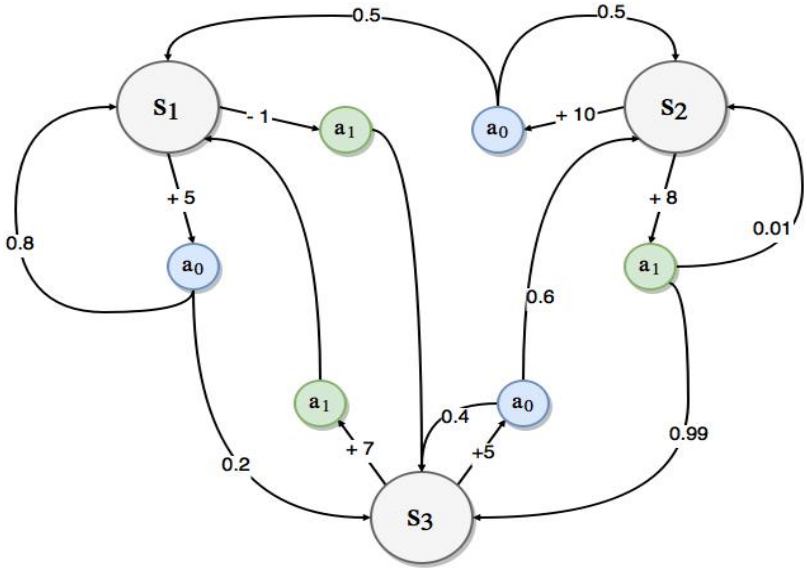


Figure 1: MDP block diagram.

One of the problems in the field of optimal control is how to formulate the safety-related specification to incorporate into the algorithms, and how to ensure that these specifications are satisfied during the exploration of the environment. The framework used to cope with this problem is the Constrained Markov decision process (CMDPs), that is a Markovian Decision Process with the addition of safety-related constraints. Indeed, the Constrained Markov Decision Process is becoming the standard for describing feasible behaviour in constrained algorithms and it can

be defined as [4]:

$$\{S, A, f, r, c\} \quad (2)$$

with the difference that in this case f is a *deterministic state transition function* such that $f: S \times A \rightarrow S$ and c is an indicator cost function such that $c: S \rightarrow [0,1]$, where 0 means safe and 1 means constraint violation. Both f and c are unknown and have to be learned from data [4]. One of the advantages of CMDPs is that there is a cost function separate from the reward function. In fact, what has always been done during the design of a reward function is to choose a trade-off between a reward for solving the task and a penalty for hazardous behaviour. This method, i.e. fixed trade-off, can lead to different problems. For example, with a too-small penalty, the robot will learn unsafe behaviour, while with a too severe penalty, the robot will fail to learn everything. Thus, separating the cost function from the reward function seems to be the most natural choice since constraints are an obvious way to formulate safety-related requirements, in contrast to the standard MDPs that just maximize the reward function, CMDPs try to make a system safe, with reference to the reduction or avoidance of harm, broadly defined, which in a practical sense means avoiding the circumstances (state and action) that are hazardous.

The mathematical formulation of the problem is described in the equation (3). Let denote Π_C as a feasible set of constraint-satisfying policies; the optimal policy in the Constrained Markov decision process framework is given by [4]:

$$\pi^* = arg\,max\, J_r(\pi) \quad (3)$$

where $\pi \in \Pi_C$ and $J_r(\pi)$ is a reward-based objective function [10]. As it has been already said, In CMDPs there is a cost function c separated from the reward function. Based on that cost function, the feasible set in a CMDP is given by:

$$\Pi_C = \{\pi : J_c(\pi) \leq d\} \quad (4)$$

where $J_c(\pi)$ is a cost-based constraint function that is similar to the reward function, and d is a threshold that is selected by a human. Through properly designed cost

functions, we are able to develop a wide range of constraints on robots' behaviour as well as any request associated with the reward hypothesis, which specifies all that we consider goals to achieve, can be defined with reward functions. Furthermore, using constraints can also improve the simplicity with which safety specifications are learned and transferred between tasks and the robustness with which agents achieve those safety-related requirements [10].

Although the CMDP is becoming a standard with regard to security problems there are still problems related to it. One of them is related to the "agent alignment" problem, i.e. the phenomenon for which the robot does not behave in agreement with the human's intentions. This problem is expressed through an issue in reward specification, therefore rewards function that in appearance seems correct, leads to unsafe and incorrect behaviour. In the same way, making a mistake in designing the cost function could lead the robot to unsafe behaviour, not solving the initial problem of agent alignment [10].

Let consider the specific case of the MPC application to Markov decision processes. MDP is used by the robot to find the actions' sequence that gives the best performance over a certain time interval, known as prediction horizon H (more details are given in the next paragraphs). In other words, it means finding the path of steps that has the highest expected accumulated reward. Therefore, the MPC steps for the Markov decision processes are the following [9]:

- ➔ Roll the horizon to the current step by observing the system state. Then, starting from the observed state, define the optimization problem of finding the actions over the control horizon that maximize the sum of the rewards;
- ➔ Determine the sequence of actions that leads to the path with the highest accumulated reward;
- ➔ Implement the first action of this sequence and move on to the next decision step.

1.2 Methodology

A model predictive control-based algorithm is used with a neural network to learn

the state transition function. The MPC provides closed-loop control which prevents cascading of errors.

The methodology can be broken down into the following major steps, as summarized in the Figure 2 [21]:

- MPC algorithm development;
- Data Collection through random actions;
- Training of Neural Network with collected data;
- Collection of random sequences of actions by random shooting;
- Evaluation of the sequence of actions through Neural Network;
- Execution of the first action of the sequence with the highest reward/lowest cost.

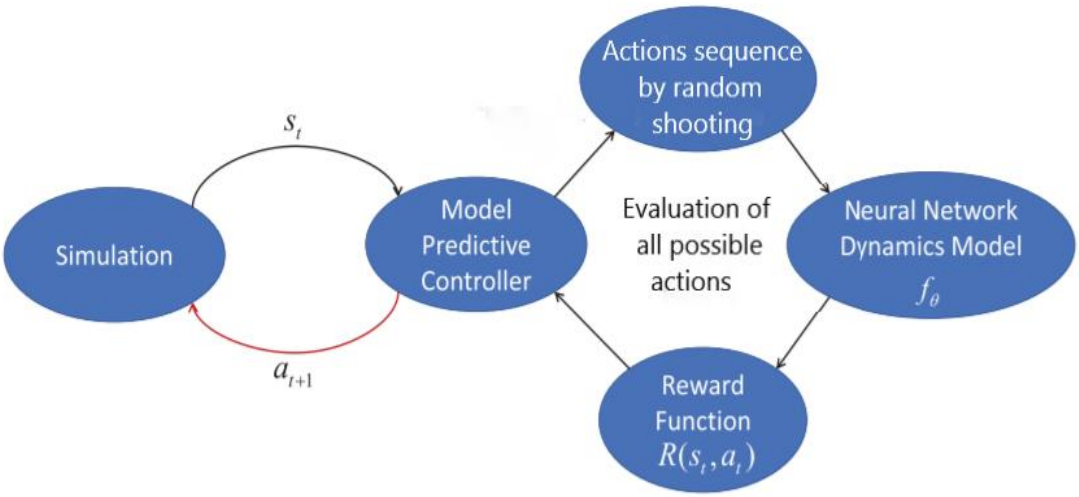


Figure 2: Algorithm Flow Chart.

The optimal control problem is studied for the tasks where environment dynamics and the analytical expression of the constraint function are unknown. A model-based predictive control algorithm for environments with continuous states and actions space is considered. In this specific case of study, the environment dynamic function is parametrized through a deep neural network $f_\theta(s_t, a_t)$, where the parameter vector θ represents the weights of the network. The DNN is used to learn the state

transition function of the robot in a given environment, where random trajectories τ are generated by providing random actions and stored in a dataset D. The trajectories are then spliced to give the current state s_t and the action a_t as input, while the predicted next state s_{t+1} would be taken as output [21]. However, the actual state and the next one might be too similar and the action has seemingly little effect on the output, thus making the transition function difficult to learn. This is true, especially for a smaller time between states Δt. To overcome this issue the considered dynamics function would predict the change in state s_t over the time step duration Δt. Thus, the predicted next state is as follows [15]:

$$s_{t+1} = s_t + f_\theta(s_t, a_t) \quad (5)$$

After learning the dynamic model of the environment, several action sequences can be generated via different methods such as Random Shooting Method or by Cross-Entropy Method. A comparison of both techniques is provided later (Section 1.4 and 1.5). For each action sequence, the dynamics model f_θ is used to predict the future states and the cumulative rewards. After the evaluation of all sequences, the optimal one is chosen and the first action is executed. The network is trained without any prior information on the required task. This approach is evaluated in the Safety Gym environment, and the results show that the proposed method is able to achieve near-optimal performance in terms of constraint violation rate and accumulated expected reward (more details about the experimental results are covered in chapter 3) [15].

1.3 Model predictive control

Model predictive control is a model-based control approach that has gained remarkable success in the process industry for different reasons, such as its ability to handle complex control problems naturally under several constraints, MPC is not a new method of control design, but essentially it solves standard optimal control problems (except that, in MPC, the optimal control problem is required to have a finite horizon in contrast to the infinite horizon usually employed in linear optimal control) [14]. Where it differs from other controllers is that it solves the optimal

control problem on-line for the current state of the system, rather than determining off-line a feedback policy (that provides the optimal control for all states). In MPC, a model is used to predict system dynamics under various robot actions. Philosophically MPC reflects human behavior whereby the control actions which will lead to the best predicted outcome (or output) over some limited horizon are selected. To make this selection an internal process model is used and constantly updates decisions as new observations become available [16]. Therefore, MPC considers the predicted behavior over a defined horizon into the future and at each next sampling instant, when new information comes available, the trajectory is automatically upgraded to take account of it [16]. Therefore, the basic idea is to use a mathematical model of the system to be controlled to predict its behaviour n time steps into the future (hence, model predictive). Of course, the prediction depends on the specific control used. Then, the best (optimal) predicted future has to be chosen. This requires a performance measure, that is, a criterion for comparing the different possible futures. Usually, this criterion takes the form of a "cost" to be minimized. Once the best sequence of controls is chosen, the first element of the sequence is applied to the system. After this first control value is applied, the system responds, a new measurement is taken, and the problem is repeated. A scheme of the general learning procedure is shown in Figure 3 [13], where the system alternates between collecting trajectories and learning a dynamics model.

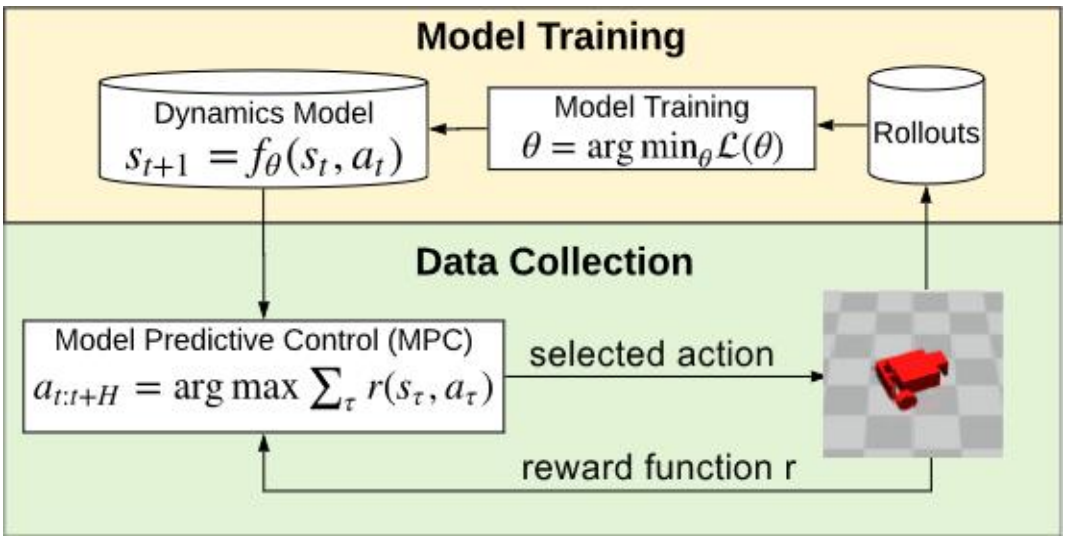


Figure 3: Overview of the general learning system on the robot.

As it has already been mentioned, modelling the dynamics of the system is a key point of the optimal control procedure, indeed, in order to predict the future behaviour of a process, it is necessary a model of how the process behaves. In particular, this model must show the dependence of the output on the current measured variable and the current/future inputs. This model does not have to be linear (e.g. transfer function, state-space) and in fact can be just about anything [16]. Most MPC algorithms use linear models since most of the time the dependence of the predictions on future control choices is linear and this facilitates optimisation. However, when linear approximations are not accurate enough, non-linear models can be used. It should be noted that, in predictive control, the model's aim is to compute system output predictions, so the model works well if it gives accurate enough predictions. The effort and detail put into the modelling stage should reflect this. There may be no need to model all the details if it is not required. The suitable model type depends on the process to be controlled. In this case, the focus is on the discrete time case, as MPC is usually implemented in discrete time [16].

1.3.1 General MPC formulation

In MPC, the robot uses a system model to predict the behaviour of a dynamic system at discrete decision steps. At each decision step, based on the system state observation and on the specific policy, the robot defines the sequence of next action that brings the system in a desired state, while minimizing negative effects of the actions, and taking constraints into account. In particular, the current control action is obtained by solving on-line, at each sampling instant, a finite horizon open-loop optimal control problem, using the current state of the system as the initial state; the optimization yields an optimal control action sequence and the first action in this sequence is applied to the system. Applying optimization in an online fashion allows the robot to deal with deviations from the plan and generate robust behaviour that reacts to changes in its environment [17]. This goal is typically specified via a deterministic reward function $r,$ so that the reward signal $r(s_t, a_t)$ encodes how desirable it is to apply the action a in state s in order to solve the control task. In particular, at every iteration, the current state of the robot is measured, and a

trajectory optimization algorithm is applied to obtain a locally-optimal state-control trajectory emanating from the current state. The initial part of this trajectory is then used as a policy while the optimization is repeated. The trajectory optimizer is warm-started with the solution from the previous iteration, which greatly speeds up the method and often yields convergence after a single optimization step [18].

More formally, the MPC discrete-time formulation can be represented by [18]:

$$x_{k+1} = f(x_k, u_k), \quad (6)$$

$$y_k = h(x_k) \quad (7)$$

Where the states x_k, controls u_k, and outputs y_k are vectors. In the sequel, x^+ is often used to denote the successor state $f(x, u)$ and thus, (6) and (7) can be written in the abbreviated form [22]:

$$x^+ = f(x, u) \quad (8)$$

$$y = h(x) \quad (9)$$

The *transition function* $f: X \times U \rightarrow X$ assigns the state $x_{k+1} \in X$ at the next time instant to each pair of state $x \in X$ and control value $u \in U$. It is learned from data and can be viewed as black-box functions [22].

Given the current output y_k, for the event (x, k) (i.e. for state x at instant k) the aim of the control is to minimize the following objective function [20]:

$$J(x, k, u) = \phi(y_{k+N|k}) + \sum_{j=0}^{N-1} L\left(y_{k+j|k}, u_{k+j|k}, \Delta u_{k+j|k}\right) \quad (10)$$

Where the double index is used to indicate values at the time $k + j$ given information up to and including instant k. Another feature of the MPC represented by the equation (10) is the term $\Delta u_{k+j|k} = u_{k+j|k} - u_{k+j-1|k}$, which offers performance advantages in that control action since it can be limited by adding constraints without introducing permanent offset in the output. The cost function L, instead, is chosen in order to maximize profit or minimize the operating costs. An

optimal sequence of controls that minimize such objective function is often indicated using an asterisk $u^*_{k+j|k}$, and in that control sequence, only the first is implemented [20]. After a new measurement becomes available, the parameters of the problem are updated and a new optimization problem arises whose solution provides the next control. Thus, repeating this optimization procedure using a dynamic objective function (i.e that is modified through process feedback) is one of the key features of MPC. The equation (10) can be solved with many optimization methods, such as random shooting (RS), and the cross-entropy method (CEM) which are explained in the next paragraphs.

1.4 Random Shooting approach

The proposed MPC problem is implemented by using the random shooting approach, which is popular in the robotics and machine learning communities due to its simplicity and its fast implementation. The method is based on generating a large number of random action sequences, followed by selecting the random sequence that is feasible (i.e., either satisfies all constraints over the whole prediction horizon or reaches the highest reward) and features the best value of the performance index. In particular it optimizes the action sequence $a_{t:t+H}$ to maximize the expected planning reward under the learned dynamics model. The robot generates K candidate random sequences of actions from a uniform distribution, and evaluates each candidate using the learned dynamics. The optimal action sequence is either approximated as the one with the highest return or the one with the lowest cost [15]. A RS agent only applies the first action from the optimal sequence and re-plans at every time-step. Even though the resulting control sequence is sub-optimal, it is still feasible, thus guaranteeing a safe operation of the controlled processes. Moreover, sub-optimality can be reduced by increasing the number of random scenarios, that is required for a satisfactory performance in particular in high-dimensional problems.

To summarize, the RS algorithm follows these three main steps [11]:

Algorithm 1 Random Shooting Algorithm

1. Generate K random control action sequences $a_t, ..., a_{t+H-1}$ picking each sequence from a uniform distribution
2. Evaluate the reward and the cost of all K action sequences by simulating environment dynamics through the DNN model
3. Pick the control action sequence that results in the lowest cost or highest reward

Algorithm 1: Random Shooting method.

1.5 Cross-Entropy Method for Optimization

To select an action, MPC searches for an optimal action sequence under the learned model and executes the first action of that sequence, discarding the remaining actions. Typically, this search is repeated after every step in the environment, to account for any prediction errors by the model and to get feedback from the environment. In many works this planning step is done using CEM. It is an efficient, derivative-free optimization method that has demonstrated good performance in optimizing neural network functions and can handle both reward and cost functions.

In the MPC setting, CEM is used every timestep to optimize an n-step planning problem on the action sequences. To compute an action plan, CEM samples a set of action sequences at each iteration, fits a normal distribution to the best samples, and samples the next population from the updated distribution in the next iteration [19]. In this way it is able to converge in a region of near-optimal solutions. More formally, it samples n dimensional actions sequence $X \in \mathbb{R}$ from a time-evolving distribution θ that is usually a Gaussian distribution $X \sim N(\theta)$, where $\theta = (\mu, \sum)$,

with mean μ and covariance matrix $\sum$. These action sequences are simulated in an open-loop fashion using the learned dynamics model to obtain approximate resulting state sequences, rewards, and costs [4]. The principle behind the CEM method is to get as close as possible to the optimal importance sampling distribution, by repeatedly sampling random action trajectories, evaluating them under the model, and re-fitting the sampling distribution to the best trajectories. The iteration's stopping criterion is often determined by a predefined maximum iteration number and a threshold on the covariance. The algorithm is the following [11]:

Algorithm 2 : Cross-entropy method

1 Pick initial mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$
2 **for** $i = 0$ **to** $n\_iterations$ **do**
3 Generate K candidate control action sequences from the mean μ and covariance Σ
4 Evaluate the reward and cost of all K action sequences by simulating environment dynamics through DNN model
5 Order the sequences, from those that result in the smallest cost to the largest, $\boldsymbol{A}_{ordered} = \begin{bmatrix} \boldsymbol{a}_1 & \cdots & \boldsymbol{a}_K \end{bmatrix}^T$
6 Choose n_elites first sequences from the ordered list $\boldsymbol{A}_{elites} = \begin{bmatrix} \boldsymbol{a}_1 & \cdots & \boldsymbol{a}_{n\_elites} \end{bmatrix}^T$
7 new mean vector $\boldsymbol{\mu} = \mathbb{E}[\boldsymbol{A}_{n\_elites}]$ and new covariance matrix $\boldsymbol{\Sigma} = cov[\boldsymbol{A}_{n\_elites}]$
8 **end**

Algorithm 2: Cross Entropy method.

1.6 Safe Exploration

In the previous sections, the optimization of the performance of a robot in a dynamical system is considered. In this context, it is also necessary to guarantee that the resulting performance (i.e., the actions that the robot performs in the environment) is safe. Indeed, an optimal model-based control approach needs to explore the environment in order to collect training data useful for the dynamic model definition, and, with regard to this, safety requirements satisfaction is crucial. Safety-related constraints are an integral part of the problem to be solved and for this reason a structure has been used that allows learning while preserving safety at the

same time. The problem of safety arises since the aim is to learn the dynamics of an unknown system from data and, based on the model, derive a policy that optimizes the long-term behavior of the system. Although for simulated environments the need for large training data sets is usually not a problem, with real robotics systems, the collection of training samples could be not only lengthy but also dangerous (both for the robot itself and for its surroundings whether it is objects or people).

However, before examining the problem of safe exploration, it is first needed to define what exactly is intended with the term *safety.* Unfortunately, in literature there is no common definition; thus, several different approaches are adopted. A possible (but vague) definition could be the following: “State-space exploration is considered safe if it doesn't lead the agent to unrecoverable and unwanted states.”, where in general the term “unwanted” does not mean low-reward states [12].

A widely used approach for safety definition is giving different labels to the states and actions indicating their level of safety. For example, a state s is called *fatal* if it is an undesired or unrecoverable state, this means that the robot is considered broken when it reaches that state. In the same way, an action is called fatal if it leads to a fatal state. Opposite to the fatal state/action, there is the *goal* state/action that corresponds to a non-critical situation. However, a novelty definition of safety divides the state space into *safe*, *critical*, and *unsafe* states [12]:

- A state is *unsafe* if it means the agent is damaged, broken, stuck, or it is highly probable that it will get to such a state regardless of further actions taken.
- A state is *critical* if there is a not negligible action leading to an unsafe state from it.
- A state is *safe* if no available action leads to an unsafe state (however, there could be an action leading to a critical state).

The definition above is better illustrated in Figure 4 (4(a) and 4(b)) [12].

(a) A safe state

(b) A critical state

Figure 4: Illustration of safe and critical states

In Figure 4(a), the UGV (Unmanned ground vehicle) is in a safe state. Indeed, whatever the action it will take, will not be dangerous since it will remain again in a safe state (supposing that actions for movement do not move the robot for more than a few centimeters). On the other hand, the robot as depicted in Figure 4(b) is in a critical state, because if it still goes forward, it will fall over and it will break [12].

Thus, safety is an important issue connected with the framework of optimal control. As the figure above suggests, there are many environments in which safety is a critical concern and certain errors are unacceptable. Just consider robotics systems that interact with humans, they should never cause injury to humans while exploring the environment in which they act. With regard to this, a safe behaviour of a robot is requested to perform exploration of unknown states. Indeed, the role of safe exploration is to provide a framework allowing exploration while preserving safety, in the sense of the existence of undesirable states or, more generally, transitions that must be avoided as they could lead to damage. In this context, the exploitation - exploration dilemma is also placed: it is necessary to find an optimal trade-off between exploitation (finding and executing optimal actions based on the current knowledge of the system) and exploration, where the system traverses regions of the state-space that are unknown or uncertain to collect data allowing to improve the estimation of the model. There are several approaches to overcome the problems where the exploratory actions may have serious consequences. These methods iteratively optimize the performance along finite-length trajectories at each time step, based on a known model that incorporates uncertainties and disturbances acting on the system (although a learned dynamics model can generally remain accurate around trajectories in the training data, its performance for unseen state-actions is not guaranteed). In a constrained MPC setting, these local trajectories are optimized under additional constraints. Safety is typically defined in terms of recursive feasibility and constraint satisfaction. Moreover, most of the optimal control is learning with no external knowledge of the task. In such cases, the exploration strategies result in the random exploration of the state and action spaces to gather knowledge on the task. This allows the controller to improve over time, given limited prior knowledge of the system. Indeed, only when enough information is

discovered from the environment, does the algorithm's behaviour improve. The randomized exploration strategies, however, waste a significant amount of time exploring irrelevant regions of the state and action spaces or lead the robot to undesirable states which may result in unwanted behaviour. In this context, one of the best simulation environments in the field of safe exploration's research is Safety Gym (more details are given in chapter 3).

CHAPTER 2

The chapter 2 is focused on a literature overview about one the two main paradigms used in the optimal control approach, i.e., model-based. Specifically, a neural network-based method has been considered for the proposed optimal control problem, where the control objective is to minimize a control-relevant cost function. Indeed, the aim of this work is to model the environment through a neural network and use the obtained model to predict the future cost over a prediction horizon which has to be minimized. In other words, in the control (MPC) algorithm, the feedforward neural network models are used to predict the state variables over a prediction horizon within the model predictive control algorithm for searching the optimal control action.

2.1 Model-based learning

The model learning's aim is to learn a transition function in order to predict the next observation given the current observation and action. This is typically done in a supervised learning approach, where a dataset with training set and test set is required [23]. Algorithms can generally be divided into Model-free (MF), also known as direct, which learn a direct relationship from states to actions and the cost is directly optimized, and Model-based (MB) ones, also known as indirect, which learn a model of the environment (in a such way it is possible to overcome the issue of a lack of prior knowledge and use it to derive a controller input). In the latter case the dataset would be collected by the data-collecting policy during the interaction

with the environment. In particular, the main difference between model-based and model-free methods is based on the usage or not of an environment model. Indeed, in model-free methods there is no model employed. This means that the rewards and the optimal actions are derived by the trial-and-error approach. Instead, in model-based methods a model of the transition dynamics is used to derive the rewards and optimal actions. Thus, the policies are optimized using the model and the optimal policy is applied at the physical system (Figure 5 [24] illustrates a model-based approach) [24].

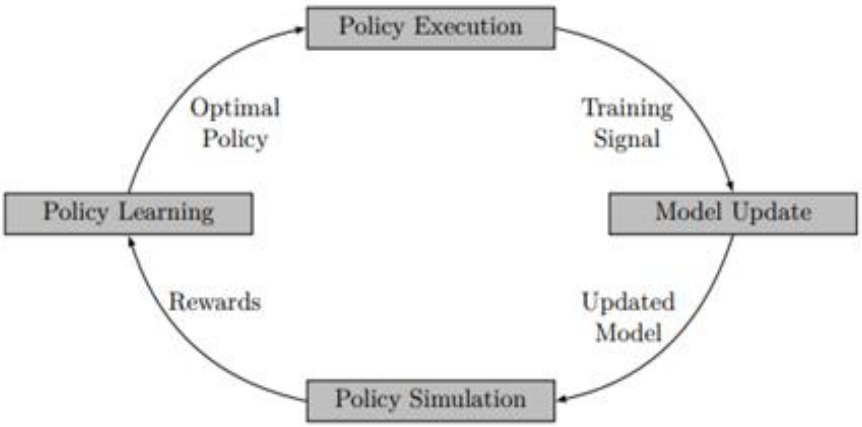


Figure 5:Pipeline of model-based algorithm.

More in detail, model-free methods use real robot interactions to create sample trajectories, i.e., the trajectories are generated by "sampling" from the robot and the system model is not required. This means that the algorithm does not know the transition from a state to another and it interacts with the next state only when the action is taken. Moreover, in spite of the relatively high number of required robot interactions for model-free policy search, learning a policy is often easier than learning accurate forward models, and, hence, model-free policy search is more widely used than model-based methods. In the model-based case, instead, the problem of sample inefficiency is addressed by using the observed trajectories to learn a forward model of the robot's environment [27]. Furthermore, in model-based algorithms, the model may be known or learned. In the former case, the agent is fully aware of the dynamic of the environment and it can use it to exploit the environment choosing the best action in that specific moment (the actions that yield

the highest reward). Indeed, knowing the model is essential to acquire the information about the next state and reward [25]. This process is known as *planning* and the most used algorithms are the ones that are based on a decision tree as illustrated in Figure 6.

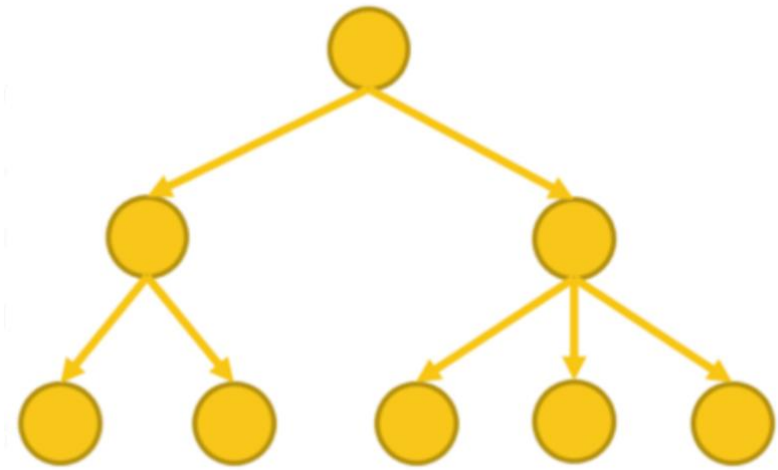


Figure 6: Decision tree.

The nodes are the states whereas all the possible actions are represented by the arrows. Each state can be reached by following a specific sequence of actions. In the latter case, a series of actions is performed, like a random or any educated actions, and the trajectory is observed. Then, a model is fit by using this sampled data. Usually, this process involves a supervised learning approach, where, in the proposed case, a deep Neural Network is trained to minimize a loss function (for example the mean squared error loss between the transitions obtained from the environment and the prediction) [25]. A more detailed description about deep Neural Networks is covered in a dedicated paragraph.

The main task of learning forward dynamics is fitting an approximation $\widetilde{f_\theta}$ of the true transition function f, where θ is the parameter vector representing the weights of neural network, given the dataset $D = \{(s, a, s_{t+1})_i\}$. Then, a policy is used to collect transition tuples $D = \{(s, a, s_{t+1})_i\}$ as the dataset for learning the dynamics, whereas a deep Neural Network allows to learn the dynamics by minimizing $\sum_i \|f(s_i, a_i) - s_{t+1_i}\|^2$ [26].

Thus, to summarize, the main steps of the basic method of learning a model using a DNN are shown as follow [26]:

1. Run base policy $\pi_0(a_t|s_t)$ (e.g, random policy) to collect $D = \{(s, a, s_{t+1})_i\}$
2. Learn dynamic model $f(s, a)$ to minimize $\sum_i \|f(s_i, a_i) - s_{t+1}\|^2$
3. Plan through $f(s, a)$ to choose actions.

Once the model is learned, it can be kept fixed or retrained overtime due to a change of the policy or a new environment's area explored. In the latter case the model is refined with additional dataset collected from new interaction coming from a different policy. This ensures the model learns all the possible parts of the environment needed to plan optimal trajectories. This operation is called *data aggregation* [25].

In practice, a fourth step should be added to the previous ones [26]:

1. Run base policy $\pi_0(a_t|s_t)$ (e.g, random policy) to collect $D = \{(s, a, s_{t+1})_i\}$
2. Learn dynamic model $f(s, a)$ to minimize $\sum_i \|f(s_i, a_i) - s_{t+1}\|^2$
3. Plan through $f(s, a)$ to choose actions
4. Execute those actions and add the new resulting data $\{(s, a, s_{t+1})_j\}$ to D.

Steps 2 and 4 are repeated to continue collecting samples and fitting the model until its performance is good enough.

There are different model learning methods to approximate the dataset, which include neural networks, and probabilistic Gaussian Process. These methods are all based on one-step prediction, i.e., given an observation and action, the model is trained to predict the next observation. However, learning a model is not trivial, since there are some potential problems to face with. Among these, the two main aspects to consider are the following [25]:

– ***Overfitting the model***: the model is too complex and overfits on a local region. In this way the global view of the environment is missing.

– ***Inaccurate model:*** the model inaccuracy leads to potential cascade errors when planning or learning a policy.

It's clear that a good model-based algorithm should deal with uncertainty problems. With regard to this, in addition to Neural Networks, there are other options, like Gaussian processes, which take into account the model uncertainty but they are very slow with large datasets. Indeed, in the case of more complex environments, which require bigger datasets, deep Neural Networks are preferred [25].

2.2 Pro and cons of dynamic models

Model-free methods attracted the most scientific interest due to their application to a range of challenging problems and has recently been extended to handle large neural network policies and value functions. This makes it possible to train policies for complex tasks with minimal features and policy, using the raw state representation directly as input to the neural network. However, due to sample complexity, especially in the case of high-dimensional function approximators, model-free algorithms' applicability to physical systems is very limited [28]. An alternative is the use of a model-based approach, that learns a model of the system with supervised learning and optimises a policy under this model. In this context, the optimal policy is derived based on internal simulations of a learned forward model that corresponds to a representation of the robot's dynamics. This characteristic significantly reduces the physical interactions between the robot and its environment.

Therefore, MB algorithms tend to have higher sample efficiency than model-free, meaning they require less data to learn a policy. In other words, by leveraging the information it's learned about its environment, model-based methods can plan rather than just react, even simulating sequences of actions without having to directly perform them in the actual environment. However, to achieve good sample efficiency, these model-based algorithms have conventionally used either relatively simple function approximators, which fail to generalize well to complex tasks, or probabilistic dynamics models such as Gaussian processes, which generalize well but have difficulty with complex and high-dimensional domains, such as systems with frictional contacts that induce discontinuous dynamics. In this context a good solution seems to be neural networks which allow excellent sample efficiency, while still being expressive enough for generalization and application to various complex and high-dimensional tasks. On the other hand, the main disadvantage is that model-based algorithms heavily depend on the model's ability to accurately represent the transition dynamics. Thus, possible model inaccuracies, when it is learned, introduce potential errors into the police, leading to a lower performance of model-based learning with respect to the model-free algorithms. Indeed, by having to learn a policy (the overall strategy to maximize the reward) as well as a model, the degree of potential error is compounded. In other words, there are two different sources of approximation error in model-based algorithms, whereas in model-free ones there's only one [29]. For similar reasons, model-based approaches tend to be far more computationally demanding, both for training the model, and for the planning operations themselves, than model-free ones, which by definition simplify the learning process. Thus, it is slower to train than model-free models. Then, the MB method requires additional memory, for example to store the model. However, with function approximation this is typically not a large limitation. It should be also noted that model-based learning is extremely useful when the model is easier to learn than the policy itself and when interactions with the environment are costly or slow.

The pros and cons of the two classes of the optimal control algorithms are summarized in Table 1 [24].

Optimal Control Methods	Advantages	Disadvantages
Model-based	- Small number of interactions between robot & environment - Faster convergence to optimal solution	-Depend on transition models -Model accuracy has a big impact on learning tasks
Model-free	-No need for prior knowledge of transitions -Easily implementable	-Slow learning convergence -High wear & tear of the robot -High risk of damage

Table 1: Advantages and disadvantages of model-free and model-based Optimal Control methods.

2.3 Neural Networks

Fast feedback control and safety guarantees are essential in modern robotics. With regard to this, the thesis presents an approach that achieves both by combining novel robust model predictive control with function approximation via deep neural networks (DNNs). Function approximation is used to represent value function through only a fixed amount of memory. The choice of the function approximator is task-dependent. The popularity of DNNs regards their efficiency and the ability to learn features by themselves. Neural networks are inspired by neuroscience, and in particular by the structure of the human brain, which performs very complex information processing (Figure 7). Indeed, the biological neurons use connections, called synapses, to receive and pass signals to other neurons. These synapses have strengths associated with them, which indicates the weight of the information that is

received from other neurons. As it can be seen from Figure 7, each neuron consists of three main components (the body, an axon, and many dendrites). Similar to the biological version, the artificial neuron is modelled as a central nucleus connected with different inputs [30].

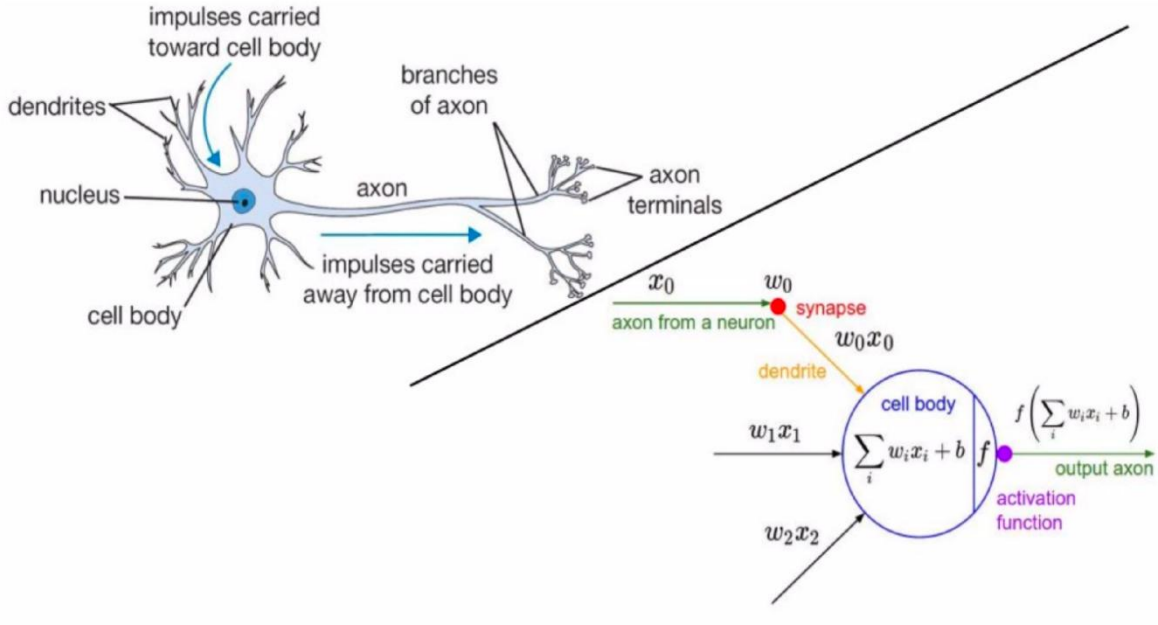


Figure 7: Neural networks and neuroscience.

The schematic representation of a basic neural network is shown in Figure 8. The architecture consists of an indefinite number of neurons organised in interconnected layers [31]: the *input layer* (leftmost layer), which represents the dataset and the initial conditions, the *output layer* (rightmost layer) and *N hidden layer* (middle layers). Indeed, the name "deep neural networks" refers to the use of many hidden layers, making it a "deep" network to learn more complex patterns. As can be seen from Figure 8 a DNN is a hierarchical organisation of neurons connected to other neurons. These neurons pass signals to other ones based on the received input and define a complex network which is able to learn with some feedback mechanism [32].

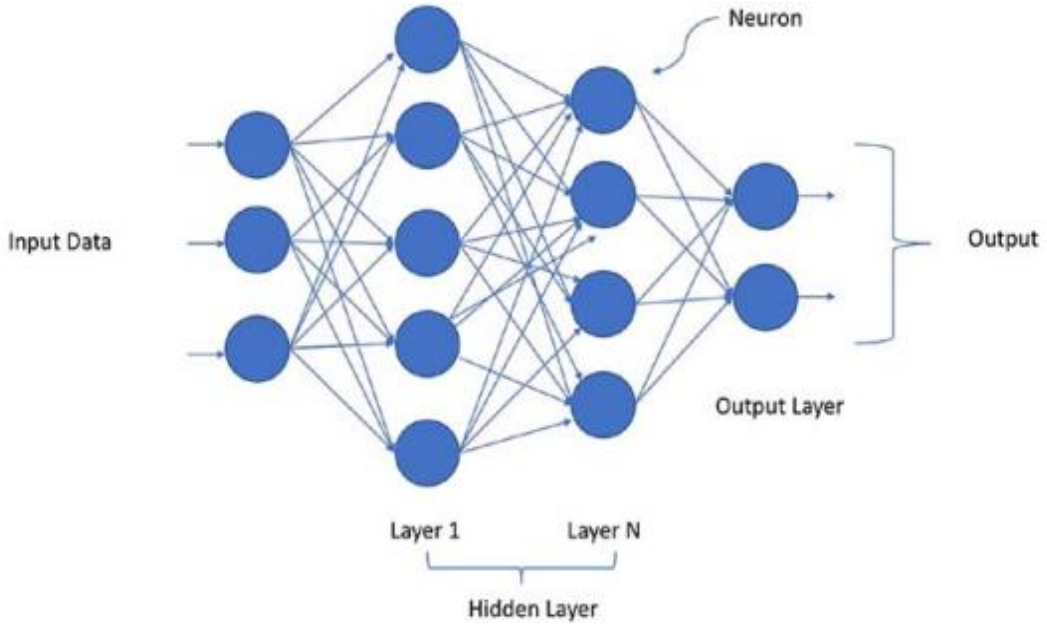


Figure 8: Example of the neural network structure.

Therefore, the core of the DNN is represented by the neurons where computation for an output is performed. A neuron receives some inputs from the other ones in the previous layer. In particular, the neurons in the first hidden layer receive the data from the input data stream, and then provide an output to the next layer and so on [32]. Figure 9 shows an example of a 3-layer *fully connected* neural network with two hidden layers. The input layer consists of k input neurons, the first hidden layer has instead n neurons, whereas the second hidden layer has m hidden neurons. The output, in this example, is made by only two nodes y_1 and y_2. On top is the always-on bias neuron [31].

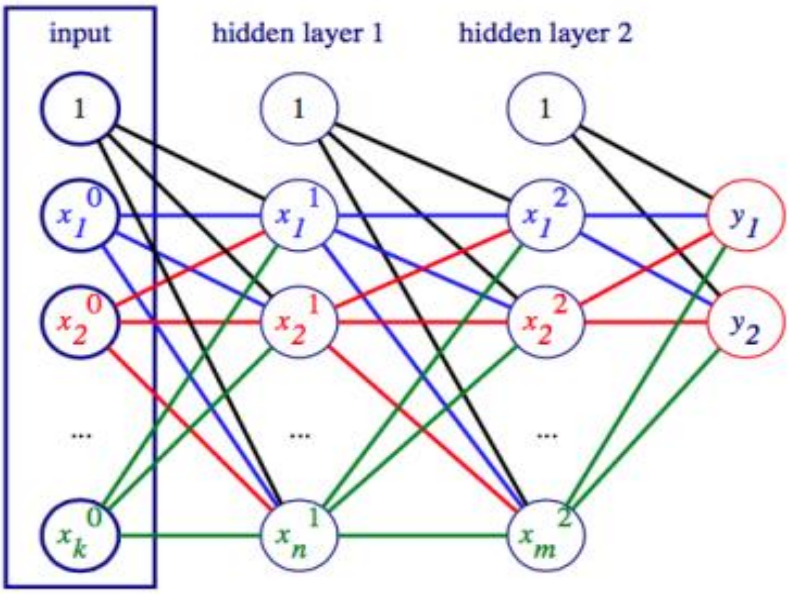


Figure 9: Multi-layer sequential network.

The connection between two neurons of successive layers would have an associated weight (in the previous case is not depicted for reasons of simplicity). Figure 10 shows the overall structure of a neural network [31]. The weights represent how the input influences the output for the next neuron and eventually for the overall final output. In a neural network, the initial weights are usually set to random values, but these weights are updated iteratively to learn to predict a correct output.

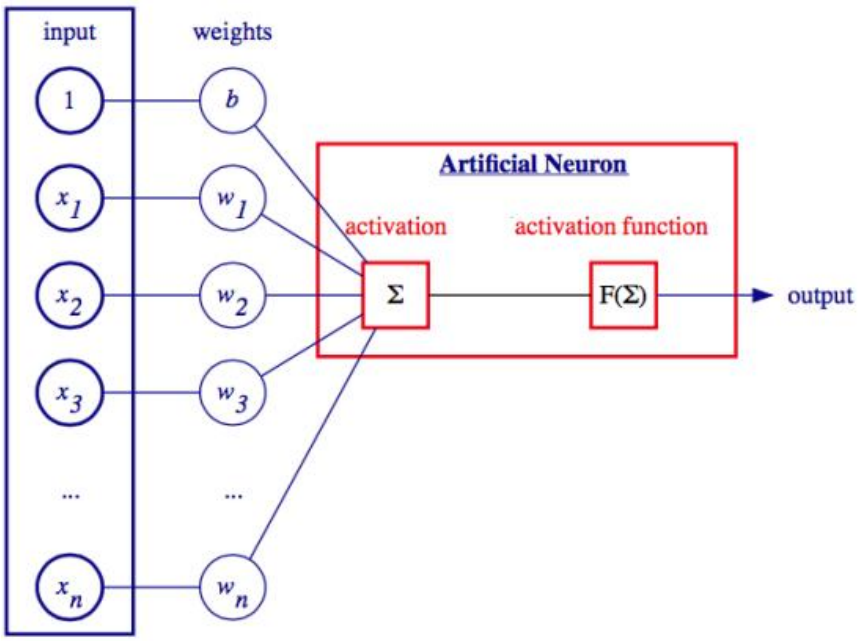


Figure 10: Different elements of the NN.

Thus, the NN is a mathematical function that maps inputs $x \in R^{nx} \rightarrow y \in R^{ny}$ via interconnected monotone functions. The mathematical model is a weighted sum of the neuron connection w_i and the respective input x_i [31]:

$$y = \sum_{i=0}^{i=N} (w_i x_i) + b \quad (11)$$

First, the weighted sum $\sum_{i=0}^{i=N} (w_i x_i)$ of the inputs x_i and the weights w_i (also called *activation value*) is computed. Here, the weight b is a special value called *bias* whose input is typically 1 [31].

Then, the result of the weighted sum is the input to the *activation function* f, which is also called *transfer function*. This function works on the computed input data as shown in Figure 11 [32].

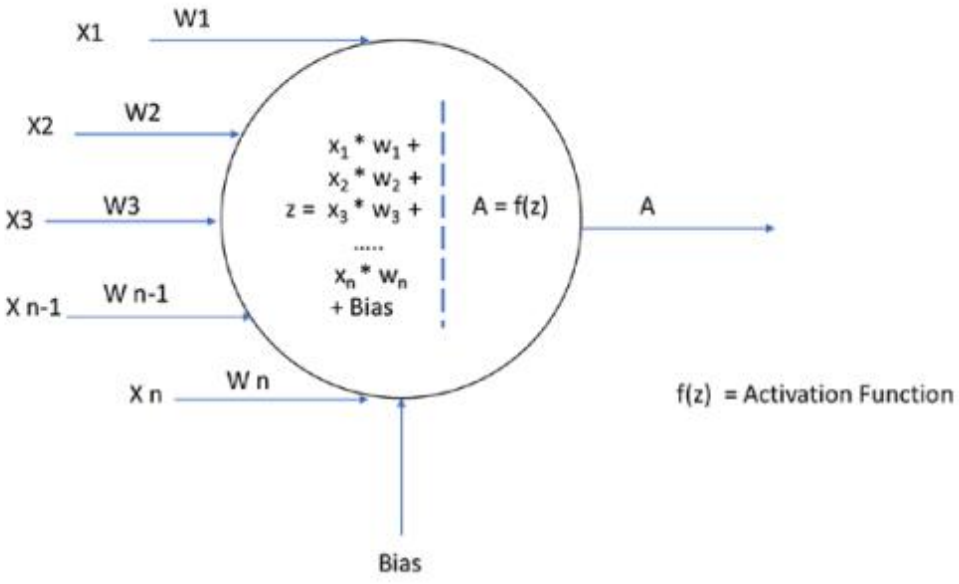


Figure 11: Single Neuron representation.

2.4 Activation Functions

As can be seen in the preceding scheme, the activation function is the function that starting from the input z, applies a function on it, and passes the output value. There are many types of activation functions, but common to all is the requirement to be non-linear. Usually, all neurons that lie in the same layer have the same activation function, but different layers may have different activation functions [32].

The specific type of activation function which is chosen can influence the quality of training of deep neural networks. The most commonly used functions are the *sigmoid function* and the *ReLU function* (rectified linear unit).

2.4.1 Sigmoid Function

The sigmoid function is expressed as [32]:

$$\sigma(z) = \frac{1}{(1+e^{-z})} \quad (12)$$

where the symbol z represents the total input to the hidden neuron. As shown in Figure 12 [32] its output is bounded between 0 and 1, which is one reason why this is commonly used for networks consisting of neurons that may activate through a probability function.

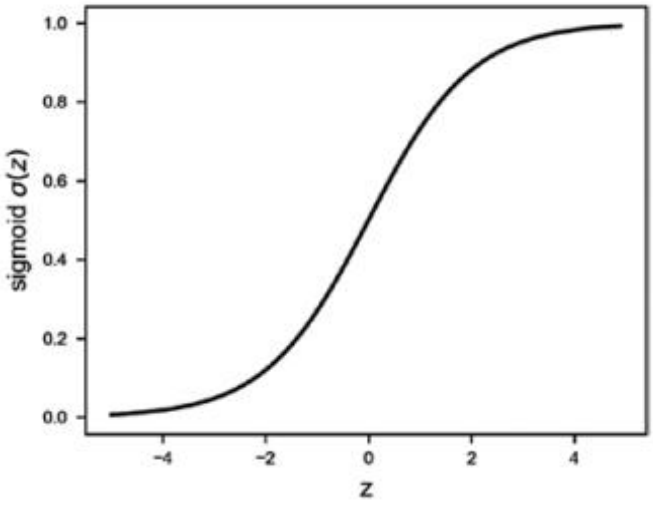


Figure 12: Sigmoid function.

Although this activation function is more convenient to use in the output layer for a probability-based output prediction, it is preferred to avoid it in the hidden layers since it leads to what is called *vanishing gradient problem*. When the value of z is either greater than 2 or less than -2, then the output of the sigmoid function is very close to 1 or 0, respectively. Because of this, the network's ability to learn has slowed down drastically [30].

2.4.2 Rectified Linear Unit Function

The ReLU is the activation function used in this work, especially for its computational efficiency which makes the network able to train a lot faster and thus to converge more quickly [30].

The ReLU function looks like [30]:

$$f(z) = max(0, z) = \begin{cases} 0 \; if \; z < 0 \\ z \; if \; z \geq 0 \end{cases} \quad (13)$$

It should be highlighted that if the output is positive, it outputs the same value, otherwise the output is 0. The function's output range is shown in the following Figure 13 [32].

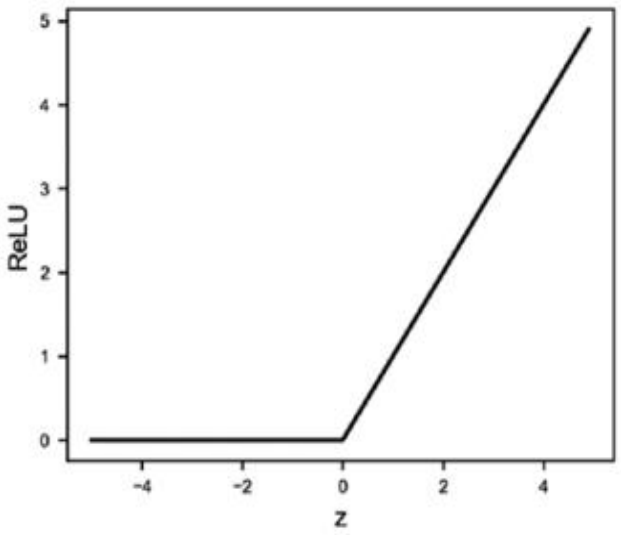


Figure 13: ReLu function.

As can be seen, all the negative values for z return into 0. The function may look linear, but it is a valid nonlinear function and in fact works really well as an activation function. It has a derivative that is as follows [30]:

$$\frac{d}{dz}f = \begin{cases} 1 \; if \; z \geq 0 \\ 0 \; otherwise \end{cases} \quad (14)$$

The derivative is shown in Figure 14 [30]:

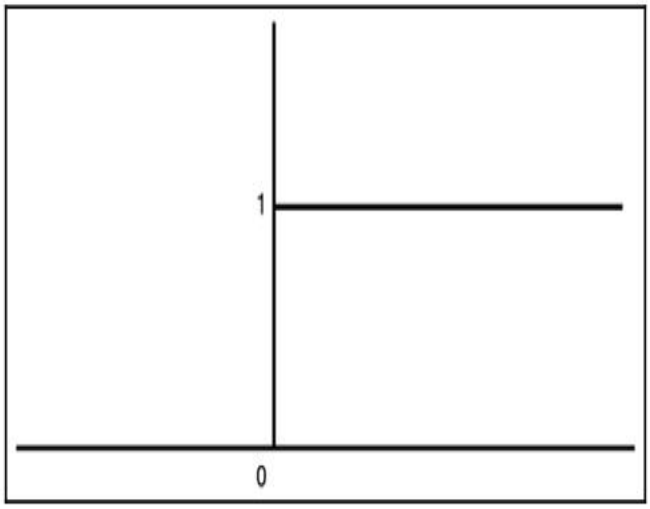


Figure 14: ReLu derivative.

This, too, has to manage some problems in the training phase, specially the d*ying ReLU problem*. This occurs for negative input values which hinds learning because it is not possible to differentiate 0 [30].

2.4.3 Leaky ReLU Function

Leaky ReLU is a modification of the ReLU function, which not only makes the network able to learn faster but it also helps deal with vanishing gradients.

The leaky ReLU function is expressed as [30]:

$$f(z) = max(0.01, z) = \begin{cases} 0.01 \; if \; z < 0 \\ z \; if \; z \geq 0 \end{cases} \quad (15)$$

The function's output range is shown in Figure 15:

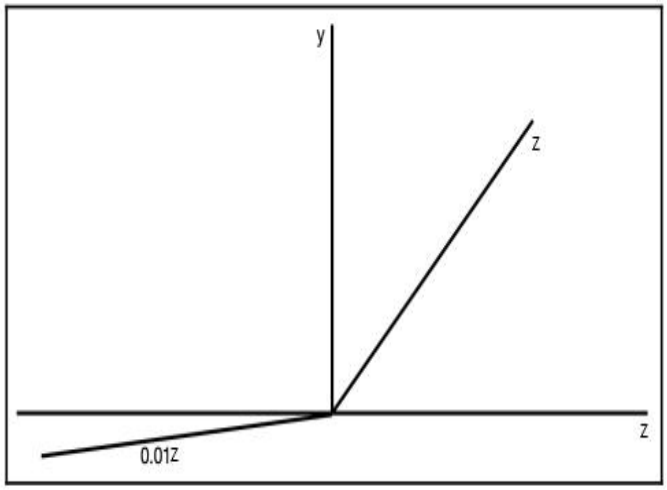


Figure 15: Leaky ReLu function.

As can be seen from the diagram above, the dying ReLU problem has been overcome by rescaling the negative value of z to $0.01z$.

2.5 Model training

To summarize, the main characteristics of a neural network are the following ones [31]:

- Neurons are the simple element where information processing occurs;
- Neurons are connected by means of connection links through which they exchange signals;
- Connection links can be stronger or weaker, and this determines how information is processed;
- Each neuron is characterized by an internal state based on all the incoming connections from other neurons;
- Each neuron has a different activation function that is computed on its state, and defines its output signal.

Now that the main concepts about neural networks are presented as well as the need to use them, it's possible to introduce the principal steps needed to train the DNN model.

They consist of:

- Getting the data ready;
- Defining the model structure;
- Training the model and making predictions.

The data given as input to the deep learning algorithm can be of different types, but they all have to be numeric data. Anyway, the model understands data as "tensors", i.e., a generic form for vectors. Data of any form is finally represented as a simple n-dimensional matrix. Once having the data ready in the necessary format, it would need to first design the structure of the DNN. Then, the number and types of layers, the number of neurons in each layer are defined as well as the required activation function, the optimizer to use, and few other network attributes. All the hyperparameters are hand-tuned to obtain reasonable performance. However, hand-tuning should not be viewed as indicative of the best possible performance.

Given the network, the training data with the correct predictions is used to train the network. Finally, the trained model is used to make predictions on the test dataset. In this context the loss function plays an important role since it helps to understand if the network is properly learning.

2.5.1 The Data

The purpose is to define a model which is able to map an input to an output. With regards to this, the network needs to be fed by lots of data. Therefore, how the data should look like represents an important key in the model training operation: each sample in the dataset consists of an input x_i and the corresponding output/target y_i. However, based on the type of requested task, the output will look a bit different, i.e., will take on any real value, in the case of a regression task, or it will be one of the predicted classes, in the case of a classification task. Anyway, the data x have to contain all the various information needed to predict the target variables y, and this, of course, is based on the specific problem [30].

The dataset is then split into three different sets, i.e., training, testing, and validation

sets, depending largely on how much data there is. In the case of deep learning, where usually there will be very large datasets, a rule of thumb is to select the dataset to train the network, and the remaining is split into two portions, the validation, and test sets. The validation set is used during training to determine how well the model fits the data, whereas the test set is used to check how well the network generalizes to unseen data [30].

Since the training of the neural network is done in a supervised learning fashion, it is needed to provide to the neural network both the input and output (the target value) data. To do this the robot performs random actions that are collected in a dataset $D = \{s_t, s_{t+1}, a_t, r, c\}$. Then, the collected data are standardized using the python's library *scikit-learn* by removing the mean and scaling them to unit variance. At last, they are divided into inputs $x = (s_{t+1}, a_{t+1})$, and outputs $y = (s_{t+1}, r, c)$ that are passed through the neural network in *batch size.* The batch size is a hyperparameter that defines the number of samples that are propagated through the network before updating the model parameters. There are some advantages of using a batch size that is smaller than the number of all samples, such as less memory consumption and a faster training phase than not using it. Another hyperparameter that is important for training the neural network is the number of *epochs.* An epoch in machine learning indicates the number of passes of the entire training dataset within the neural network. Determining how many epochs a model should run to train is based on many parameters related to both the data itself and the goal of the model.

2.5.2 DNN model structure

The base class used for developing deep neural networks from scratch is *nn.Module* coming from PyTorch (more detail in Section 3.2). It can be used as the foundation to be inherited by model class since it uses tensors (data structure used by machine learning systems) and automatic differentiation modules for training and building layers such as input, hidden, and output layers.

Along with nn.Module class, *nn.Linear* is the module used to create a single layer feed forward network with n inputs ed m outputs (Figure 16).

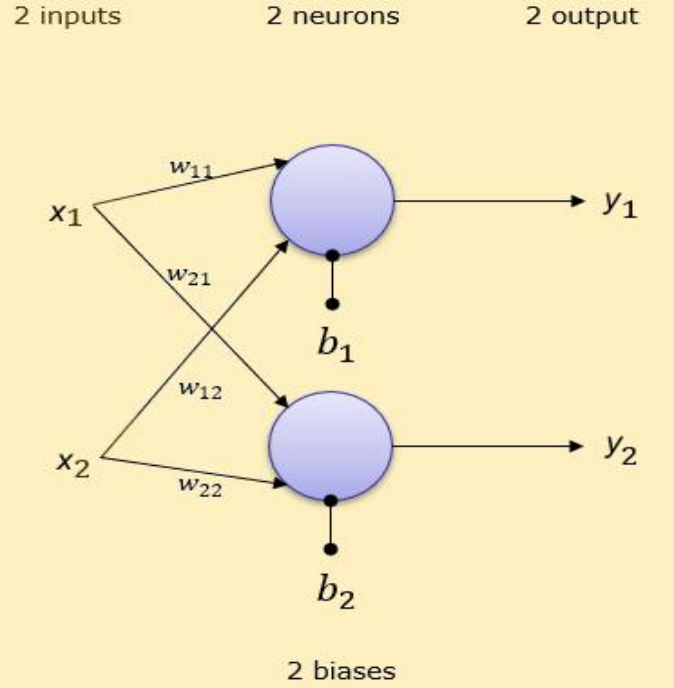


Figure 16: Example of Feed Forward Layer with 2 inputs and 2 outputs created with nn.Linear module.

Mathematically speaking, this module is designed to apply a linear transformation to the incoming data

$$y = xA^T + b \quad (21)$$

Where x is the input, y is the output, b is the bias, and A^T is the weight's matrix. The equation (21) represents one of the fundamental statistical and machine learning techniques: *Linear Regression*. Linear Regression is used to search for relationships among variables, i.e a function that maps some features or variables to others sufficiently well. In other words, the goal is to build a system that can take a vector of features $x \in \mathbb{R}$ as input and predict the value of a scalar $y \in \mathbb{R}$ as output. The dependent variables are the outputs, i.e the future states, the rewards, and costs, while the independent variables are the inputs, i.e the actual state and the action performed in that state.

2.5.3 The Loss Function

A very critical part of neural networks training is represented by the loss function which is involved in the estimation of the error of a network after a forward pass has been computed. Indeed, in training phase, the neural network predicts the output $\hat{y}$. This predicted output is used along with the target value y to compute the loss function. Two different types of error can be distinguished: the *local error* and the *global error*. The first one defines the difference between the expected output of a neuron and its current output, whereas the global one represents the total error (the sum of all the local errors) and is an estimate of how well the network is performing on the training data [30].

The loss function essentially measures the gap from the target, i.e it tells how good is the performance of the neural network with respect to the task. Based on the type of data outcome, it is possible to distinguish different standard loss functions in deep learning. The most available ones are [32]:

- *Mean Squared Error (MSE)*: the average squared difference between the actual output and the expected one. The squared difference makes it easy to penalize the model more for higher value coming from the difference. The mathematical formula is [32]:

$$MSE = \frac{1}{N}\sum_i \|\hat{y}_i - y_i\|^2 \quad (16)$$

  which is the square of the L2 norm and N is the number of samples in the training dataset. Intuitively, the error is 0 when the actual and the predicted values are equal, and the higher the distance between the points, the larger the error. The MSE always outputs a positive value and by squaring the distance between the predicted output and expected one, it allows to easily differentiate between small and large errors and thus correct them [30].

- *Mean Absolute Error (MAE)*: the average absolute error between the actual and predicted output [32]. The mathematical equivalent looks like [30]:

$$MAE = \frac{\sum_i |\hat{y}_i - y_i|}{N} \quad (17)$$

- *Root Mean Squared Error (RMSE)*: the square root of the previous MSE function and it is as follows [30]:

$$RMSE = \sqrt{\frac{\sum_i \|\hat{y}_i - y_i\|^2}{N}} \quad (18)$$

  It gives a better idea of the error with respect to the targets, by scaling back the MSE function to the scale it was originally at before the errors are squared.

- *Huber Loss:*

$$Huber\ loss = \begin{cases} \frac{1}{2}(y - \hat{y})^2\ when\ |y - \hat{y}| \leq \epsilon \\ \epsilon\,|y - \hat{y}| - \frac{\varepsilon^2}{2}\ otherwise \end{cases} \quad (19)$$

  Where ϵ is a constant term to be configured. For very low values of ϵ, the loss function is not influenced by large errors, whereas for very high values of ϵ, the loss function becomes very sensitive to large errors. It should be also noticed that when ε is very small, the Huber loss is similar to MAE, but when it is very large, it is similar to MSE [30].

Therefore, the structure needs to be defined by selecting the sequence of layers with the number of neurons, the activation functions, then, the input and output shape is initialized with random weights in the beginning, which are finally updated during the learning process by the network. At this point, the network takes one or more training samples and uses their values as inputs to the first layer, which then gives an output based on the defined activation function. The output now becomes the input for the next layer, and so on. The final layer output would be the prediction for the considered training samples. In this context the loss function is used to tune the

parameters of the network to make it perform better [32]. At this point, it is necessary to reduce the loss by acting in the proper manner on the weights thanks to an optimizer function, which is the most important part of the model training. It is a mathematical algorithm that makes a small change on the weight parameters to improve the end prediction by reducing the loss function. Step by step, with a lot of iterations, the network updates its weights and learns how to make a good prediction for the given training sample [32].

2.5.4 Optimizers

There are many popular optimizers which can be considered for different deep learning models. Among these, one of the most used is the Stochastic Gradient Descent (SGD). At each training sample, it computes the loss and updates the weight. However, the global loss curve might be very noisy due to the high frequency in updating the weights. Nowadays, the most popular and widely used optimization technique in deep learning is ADAM, which stands for Adaptive Moment Estimation. Adam can be viewed as a combination of RMSprop (RMS) and Stochastic Gradient Descent with momentum (SGD), which are two popular optimization algorithms. It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum. To put it simply, the key point of using the Adam optimization algorithm is that it is an adaptive learning rate method, which means, it computes individual learning rates for different parameters to converge faster [33]. Therefore, it differs from classical stochastic gradient descent, which maintains a single learning rate for all network weight (parameter) updates and the learning rate does not change during training. This is the learning rate optimization algorithm, specifically designed for training deep neural networks, used in this work. The *learning rate* is a hyperparameter that determines the step size at each iteration while moving towards minima in the loss function. Choosing the learning rate is challenging: a too-small value may result in a long training process that could get stuck, whereas a too-large value may result in learning a sub-optimal set of weights too fast or an unstable training process [33].

2.5.5 Parameter initialization

Once the model is defined, it is necessary to set its parameters. This is an important key point since neural networks have a huge amount of parameters to initialise. Therefore, finding the correct values that minimizes the loss is not trivial and can be very time consuming and challenging. For this reason, a good starting point makes it easier to get the optima and reduce the training time. There are various ways for initializing weights and biases [30]:

- *All zeros*: the initial weights and biases of the model are set to zero. The main drawback is that all the information about the training data and the network are lost. This is a very problem to avoid when training a neural network;
- *Random initialization*: two different distributions, the normal distribution or the uniform distribution, can be used. To initialize model parameters using the normal distribution, the mean and the standard deviation have to be defined. Usually, it is chosen with a mean of 0 and a standard deviation of 1. To initialize using the uniform distribution, instead, it is usually used the range [-1, 1];
- *Xavier initialization*: the weights are initialized just right in order to avoid the phenomenons of *exploding* or *vanishing gradients*. The idea is to maintain the variance as propagating through subsequent layers. It follows the mathematical formula

$$W_{i,j}^{|k|} \sim U\left[-\frac{\sqrt{6}}{n_k+n_{k-1}}, \frac{\sqrt{6}}{n_k+n_{k-1}}\right] \quad (20)$$

  where n_k is the number of neurons in layer k.

  In this specific case both the weights and bias are initialized from a uniform distribution $(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{n° \, input-features}$.

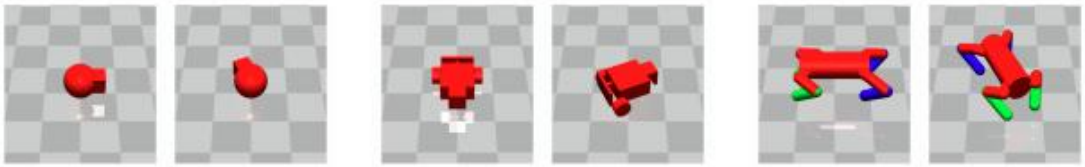
CHAPTER 3

Simulation results for investigating the performance of the MPC algorithm with deep neural network function approximators, under safety constraints, are discussed in chapter 3. The simulations are carried out in a Safety Gym environment according to task performance and safety specifications required to achieve safe exploration. The algorithm is coded using Python programming language and is evaluated based on task performance and constraint satisfaction of the final policy. Instead, the neural network used for learning the dynamics of the environment is modeled with PyTorch, a standard framework used in machine learning applications.

3.1 Simulation Environment

Safety Gym is a suite of environments and tools for measuring progress towards optimal control problems that respect safety constraints while training and it is implemented as a standalone module based on OpenAI Gym and the MuJoCo physics simulator. Its environments are inspired by the common problems that arise in robotics even if they are not the exact simulation of them. In addition, Safety Gym is highly extensible, it integrates perfectly with ROS (Robotic Operating System), thus facilitating the development of safe robotic applications. It is the most suitable simulation environment since it has separate objectives for task performance and safety that are expressed via a reward function and a cost function, and also because there is a gradient of difficulty across the environments. One of the strengths of Safety Gym is the customization of the environment: the simulator is equipped with an environment-builder that allows the creation of a new environment by mixing robots, physics elements, goals, and safety constraints. Moreover, in addition to all these elements provided by Safety Gym itself, the user can upload his personal robot's and layouts' configurations by passing the XML file in the Safety Gym's config function [10].

The pre-made robots are essentially three: Point, Car, and Doggo, as can be seen in Figure 17 [10].



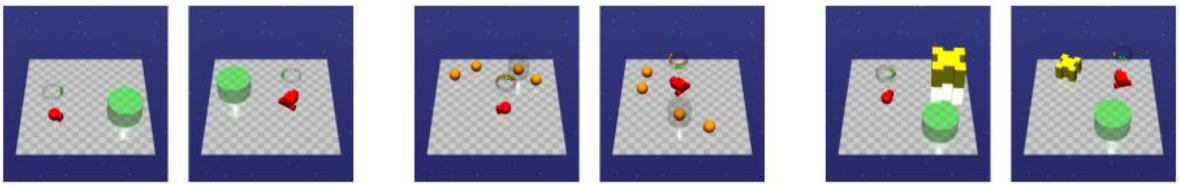
(a) *(b)* *(c)*

Figure 17: Pre-made robots in Safety Gym: (a)Point (b)Car (c)Doggo.

a) **Point:** The simplest Safety Gym's robot and the easiest to control. It is constrained to the 2D-plane and it is composed of two actuators, one for turning and another for moving forward and backward. Point has a square that's used for pushing physics elements and for determining its direction [10].
b) **Car:** Car is a robot that is composed of two independently-driven parallel wheels and a free rolling rear wheel. Compared to the Point robot, Car is a bit more complex: in addition to the different structure with respect to the Point robot, it is also not fixed to the 2D-plane, even if it resides mostly on it [10].
c) **Doggo:** Doggo is the most complex robot among the three. It is a quadrupedal robot with bilateral symmetry. It has four legs and for each of them there are two controls at the hip, for azimuth and elevation relative to the torso, and one in the knee, used for controlling angle [10]

In order to improve learning with neural networks, all actions for all robots are continuous and linearly scaled to [-1, +1].

Similarly to robots, Safety Gym has three main tasks: Goal, Button, and Push (Figure 18). For each of the tasks the reward function can be either sparse (reward obtained only when the task is completed) or dense (rewards are collected throughout the task) [10].

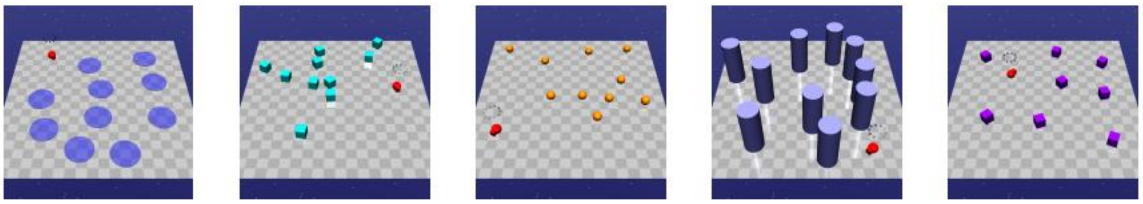


(a)Position *(b) Button* *(c) Push*

Figure 18: Tasks for the environment.

a) **Goal**: The aim is to move the robot into a series of goal positions. The sparse reward is obtained when the robot enters in the goal position while the dense reward is obtained when the robot moves towards the goal position. After the goal is reached, the position is randomly reset to a new place [10].
b) **Button:** Button is a slightly complex task with respect to Goal. There are several buttons scattered in the environment and what the robot has to do is navigate to them and press the one that is highlighted. Once the button has been pressed, the environment will select and highlight a new one. As well as the goal task, the sparse reward is obtained when the robot presses the right button while the dense reward is obtained when the robot moves towards the goal button [10].
c) **Push:** The last and the most complex task. Push consists of moving a box in a series of goal positions. Like the previous tasks, a new location is randomly selected when the goal is achieved. Reward works slightly different from the other two tasks. The sparse reward is obtained when the yellow box enters the goal circle, while the dense reward is composed of two parts: one for the robot moving towards the box and the other for pushing the box towards the goal [10].

About the safety requirements, Safety Gym supports five main kinds of elements which are shown in Figure 19: Hazards (a), Vases (b), Pillars (c), Buttons (d), and Gremlins (e) [10].



(a) *(b)* *(c)* *(d)* *(e)*

Figure 19: Constraint elements used in our environments.

These elements can be mixed and sorted freely and the user can decide for each of them whether the robot needs to satisfy a constraint or not. At every timestep, the environment provides a cost signal for each of the elements that has an associated constraint, and an overall cost signal which takes into account the overall interaction with unsafe elements.

The elements are [10]:

a) **Hazards:** Hazards are dangerous areas to avoid. The robot is penalized for entering inside the circle on the ground.
b) **Vases:** Vases are small blocks that represent fragile objects to avoid. The robot is penalized for touching them.
c) **Pillars:** Pillars are rigid and immobile barriers in the environment. The robot is penalized for touching them.
d) **Buttons:** Buttons are incorrect goals. The robot is penalized for pressing the incorrect button.
e) **Gremlins:** Gremlins are moving objects: The robot has to stay out of their path in order to avoid being penalized.

Since all the elements are different, the challenges they pose are different too. The dynamics to which the robot is subject are different according to the constraints used. Robots are equipped with sensors that allow them to detect obstacles: in addition to the classic sensors like accelerometer, gyroscope, magnetometer, and velocimeter, they have *lidar* sensors, where each lidar perceives objects of a single kind. Lidar is a method for determining ranges (variable distance) by targeting an object with a

laser and measuring the time for the reflected light to return to the receiver. In Safety Gym lidar can be Natural-Lidar or Pseudo-Lidar. Natural-Lidar is computed using ray-tracing tools, while Pseudo-Lidar is computed by looping over objects and filling bins with appropriate values [10].

3.2 Python and PyTorch

Once having a better idea of what a DNN is (Chapter 2) it's possible to jump straight into implementing a DNN using one of the most popular programming languages: *Python*. Python is a multi-paradigm programming language and it supports different programming approaches. One of the popular approaches, that's the one used in this work, is known as *Object-Oriented Programming (OOP).* It relies on the concept of *classes* and *objects*. One of the aspects that makes Python such a popular language in the field of Artificial Intelligence, is its abundance of libraries and frameworks that facilitate coding. In terms of deep learning and deep neural networks, one of the most famous frameworks, which is also the one used in this thesis project, is *PyTorch*. Before diving into its implementation, let's understand what PyTorch is and why it has become so popular. PyTorch is a Python-based scientific computing package that has at its core the added power of graphics processing units (GPUs). It provides maximum flexibility and speed during implementing and building deep neural network architectures. There are two main characteristics of PyTorch that distinguish it from other deep learning frameworks: Imperative Programming and Dynamic Computation Graphing. Imperative programming refers to the ability of PyTorch to perform computations as it goes through each line of the written code. This is quite similar to how Python programs are executed and it's a huge advantage, especially with regard to debugging. Dynamic Computation Graphing, instead, refers to the ability to generate during run time the computational graph structure of the neural network [34].

3.3 Algorithm Details

The Algorithm proposed in this work is the DNN model-based MPC one. It combines the strength of DNN in learning the model of the environment with the ability of the MPC to optimize trajectories. It is worth remembering that the dynamics model of the environment (deterministic state transition function) the cost function, and the reward function are unknown and need to be learned from the collected data. The entire pipeline of how MPC with RS works is the following:

Algorithm 3 MPC with RS

for iter=1 **to** max iter **do**
  gather dataset D of trajectories
  Train the dynamic model f_θ using D
  for Time $t = 0$ to EpisodeLength **do**
    Observe state s_t from the environment
    Optimize actions by RS method:
    $\{a_i^*\}_{i=t}^{t+T} \leftarrow$ RS
    Use f_θ to estimate optimal action sequence
    Apply the first action a_t^* in $\{a_i^*\}_{i=t}^{t+H}$ to the system
    Observe next state s_{t+1} and $r(s_{t+1}), c(s_{t+1})$
  end for
end for

Algorithm 3: MPC with Random Shooting.

The objective of MPC is to minimize the accumulated cost with respect to a sequence of actions over the planning horizon. RS generates K sequences of possible actions that are evaluated through the DNN dynamic model. After the evaluation, the first action coming from the sequence with the lowest cost is applied to the system, new observations are received, and the same optimization is performed again. Taking a step back and going deeper into the dynamic model, the key points to take into account for the training of the network are now explained. First of all, at every iteration, the dataset is collected by the robot interacting with

the environment through 200 episodes, and for each episode, the robot can perform 1000 steps during which it collects experiences (for each step) made by $\{s_t, s_{t+1}, a_t, r, c\}$. Notice that, only in the first iteration the dataset is collected by performing random actions in the environment, while in the other iterations the dataset is collected using the MPC. Once the dataset is collected, it needs to be standardized (a pre-processing technique) using python's library scikit-learn by removing the mean and scaling them to unit variance. At last, the dataset is split into train-set and test-set according to the 80-20 rule. Given the train-set, the dynamic model is trained in a supervised-learning fashion by minimizing the MSE loss between the predicted output and the real one. Usually, the point of interest is how well the model performs on data that it has not seen before, since this determines how well it will work when deployed in the real world. For that reason, the performance is evaluated using a test set of data, i.e. a set of data that is separated from the one used for training the algorithm (more details in section 3.5). It is based on these learning curves that the parameters, both of the DNN and MPC, are tuned to improve the performance.

3.4 Parameters Consideration

As has already been said, all the parameters and hyperparameters of the DNN and MPC are hand-tuned based on the environment and the task to obtain reasonable performance. However, hand-tuning should not be viewed as indicative of the best possible performance. When discussing the DNN dynamic model, the first and the most important parameter is the size of the network architecture. With that being said: the process of deciding on a network architecture largely resembles trial-and-error. However, some key considerations can be done. The width of the input and output layer is chosen based on the environment and task's dimension. The larger the dimension of the state and action vectors, the bigger is the width of the input and output layer. In the CarGoal1 task, the state's vector (i.e the actual observation which the robot receives from the environment) is composed by the measurement of 89 sensors (accelerometer, velocemeter, lidar and so on), while the action's vector is

composed of 2 actions (one for moving forward/backward and one for steering the free wheel). For this reason, the width of the input layer is 91 (89+2), as well as the width of the output layer, since it predicts the next state, the reward, and cost (89+1+1) through linear regression. The depth of the network (hidden layer), as well as the internal width, are chosen to take into account the complexity of the task to solve along with the computational cost (time needed to train). After several tests, 3 layers of 500 neurons each are chosen. Given the network's structure, the MSE loss function with Adam optimization is chosen. Even in this case, batch size, learning rate, and epoch are chosen empirically based on the performance of the algorithm and they are respectively 128, 0.0001, and 100, while the other parameters required from Adam (i.e. such as the exponential decay rate) are set to their default value. Regarding MPC instead, the parameters to be tuned are the horizon and the number of action sequences to generate over that horizon. Since the environment is stochastic, their choice is crucial for the performance of the algorithm: a longer horizon makes the MPC vulnerable to drifting since the state space is too big to be fully covered and, without a proper model, it may land in areas not learned yet. Then, if it is true that a bigger number of action sequences can increment the performance of the MPC, it is also true that it requires a large computational cost. Bearing in mind this, the horizon is set to 5 and the action sequences generated are 100, this leading to the final configuration of the MPC - RS algorithm that is evaluated in the next section.

3.5 Simulation Results

To evaluate the MPC - RS algorithm, the Safety Gym environment the Goal1 Task is considered. The experiment involves the Car Robot (red object in Figure 20) that must navigate in a cluttered environment to perform a specific task while avoiding the sparse obstacles in the environment.

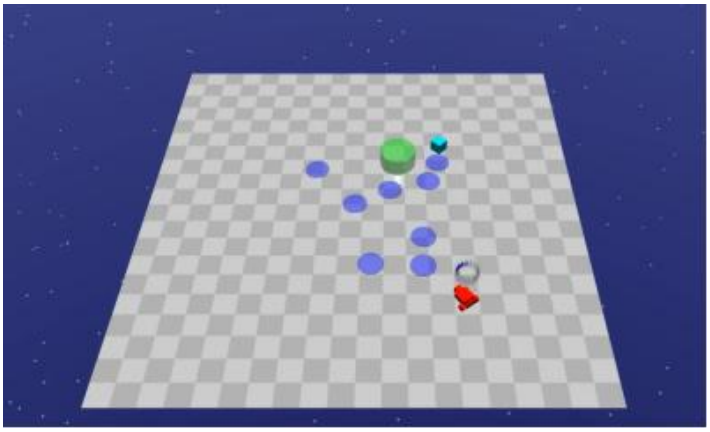


Figure 20: Car Goal1 Task.

If the robot reaches the green area a bonus of $r_t = 1$ is earned, while a cost $c_t = 1$ is associated with the violation of safety constraints that corresponds to the blue area [4]. It's worth noticing that the environment is stochastic, i.e. there is extensive layout randomization so that robots are required to generalize to safely navigate and solve tasks. In particular, the layout is randomized at the start of each new episode and also when a goal is achieved.

The baseline used to compare the performance of the MPC algorithm is Trust Region Policy Optimization (TRPO) [35]. TRPO is an uncostrained model-free policy gradient algorithm that avoids changing the policy too much during the update. To ensure that the policy does not change too much, a constraint on the optimization problem is added so that the update lies within a trust region. The constraint is expressed in terms of KL-Divergence, a measure of distance between probability distributions.

To assess the task and safety performance of the algorithms the episodic sum of returns J_r and the episodic sum of costs J_c are considered. The algorithms are also

compared based on their sample efficiency, i.e. the number of total steps needed to converge.

The following figures 21-22 show the episodic accumulated reward and the episodic accumulated cost for the TRPO algorithm:

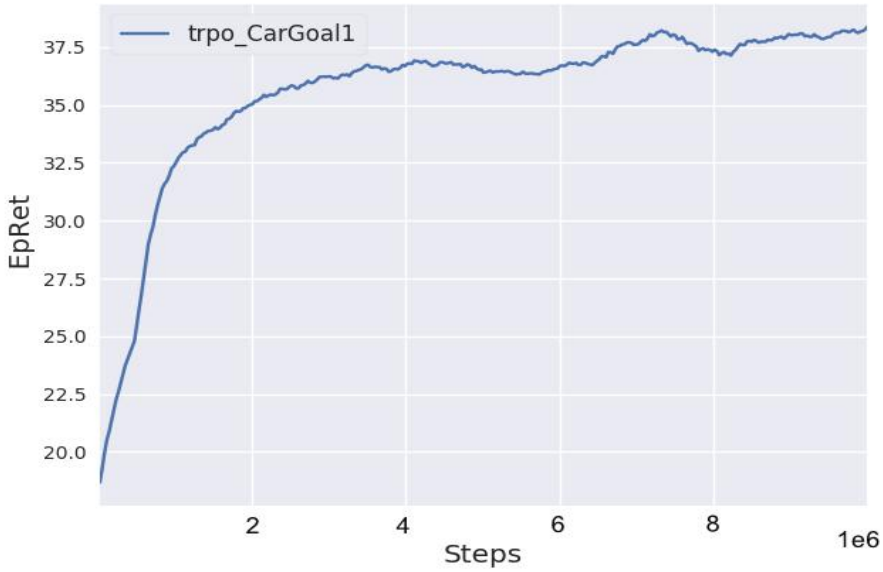


Figure 21:TRPO episode accumulated return.

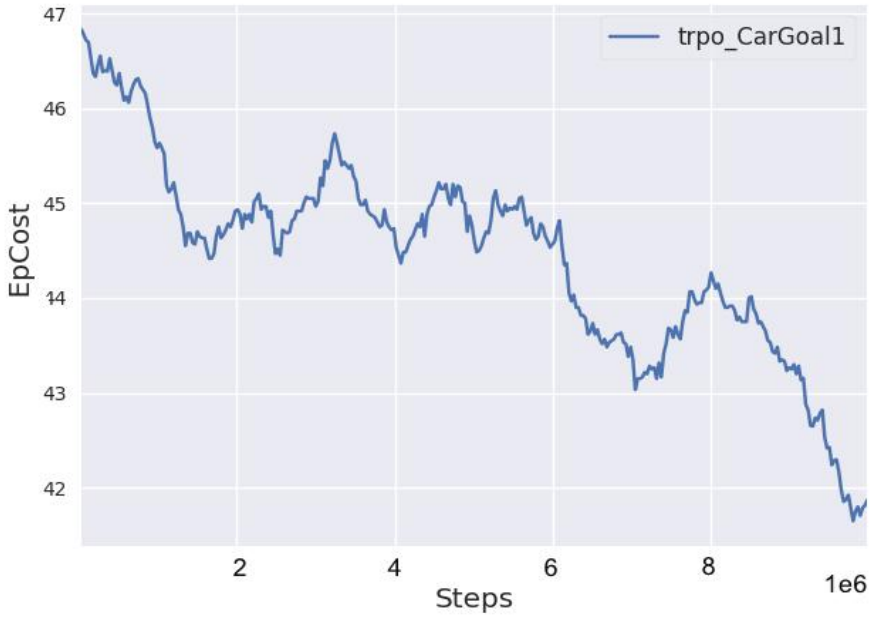


Figure 22: TRPO episodic accumulated cost.

The TRPO gives an intuition about the performance and constraints violation that can be reached when only the reward is considered. As it can also be seen, the

TRPO algorithms require millions of steps to reach convergence on performance while ignoring safety requirements.

Listed below there are some graphs and considerations to better understand how the MPC - RS algorithm behaves under certain conditions. The metrics used to evaluate its performance are essentially two: the training loss and the validation loss. Based on the combination of the two it is possible to decide how to tune the parameters and what is the best configuration to compare to TRPO.

It is reported here on of the first training performed

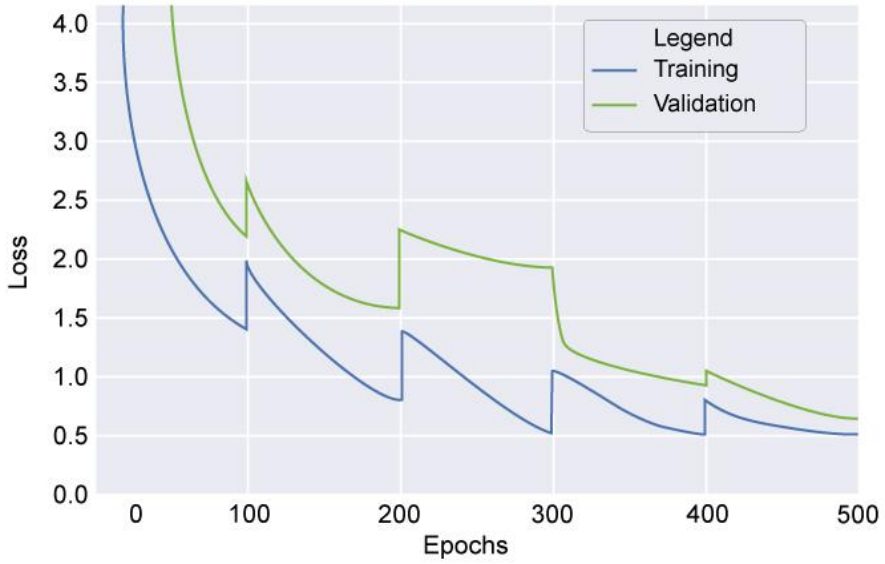


Figure 23: Training and validation Loss before using replay buffer.

As can be seen, the training loss and the validation loss have an unexpected behavior because of their discontinuity at the beginning of each iteration. This may be caused by a memory loss about data that the algorithm has already seen, thus to overcome this issue a replay data buffer is implemented. In other words, in each iteration the data collected after the interaction with the environment are stored in a buffer which contains also the data of the previous iteration (this technique is called *Data Aggregation*). Therefore, the replay buffer is used as input for training the dynamic model at each iteration. The algorithm 3 becomes as below:

Algorithm 4 MPC with RS

Initialize $D_{RB} = 0$

for iter=1 **to** max iter **do**
  gather dataset D of trajectories
  Train the dynamic model f_θ using $D + D_{RB}$
  for Time $t = 0$ to EpisodeLength **do**
    Observe state s_t from the environment
    Optimize actions by RS method:
    $\{a_i^*\}_{i=t}^{t+H} \leftarrow$ RS
    Use f_θ to estimate optimal action sequence
    Apply the first action a_t^* in $\{a_i^*\}_{i=t}^{t+H}$ to the system
    Observe next state s_{t+1} and $r(s_{t+1}), c(s_{t+1})$
    Update data buffer:
    $D_{RB} \leftarrow D_{RB} \cup \{s_t, a_t, s_{t+1}, r(s_{t+1}), c(s_{t+1})\}$
  end for
end for

Algorithm 4: MPC with Random Shooting and Data Buffer.

As it can be seen, as opposed to the algoritm 3 shown in Section 3.3, a new dataset D_{RB} is implemented to store all the data. That change impacts significantly the training of the dynamic model shifting the focus on other parameters.

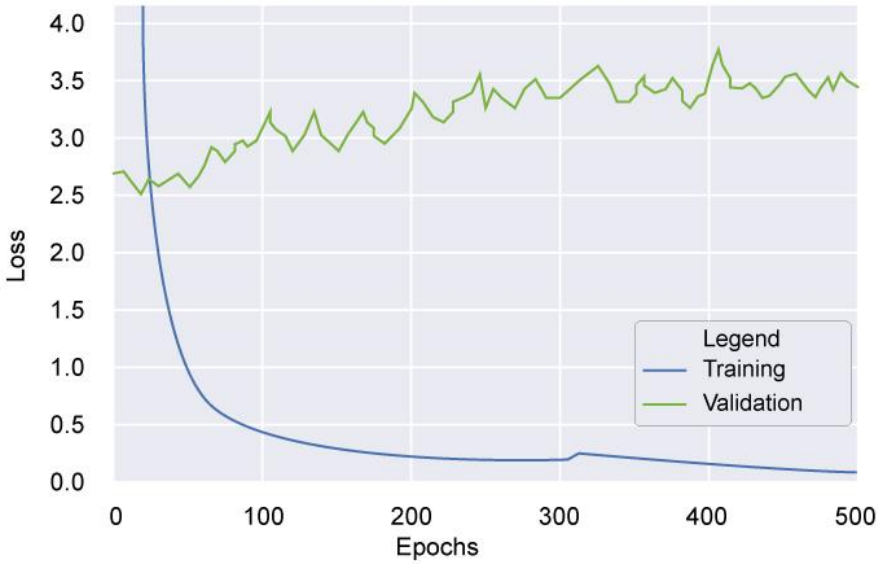


Figure 24: Behavior of training and validation loss when the model is overfitted.

Figure 24 shows the behavior after the implementation of the replay buffer. Looking at the training loss and validation loss, it is clear that the model is overfitted, i.e. the model is more accurate in fitting known data then the unknown one.

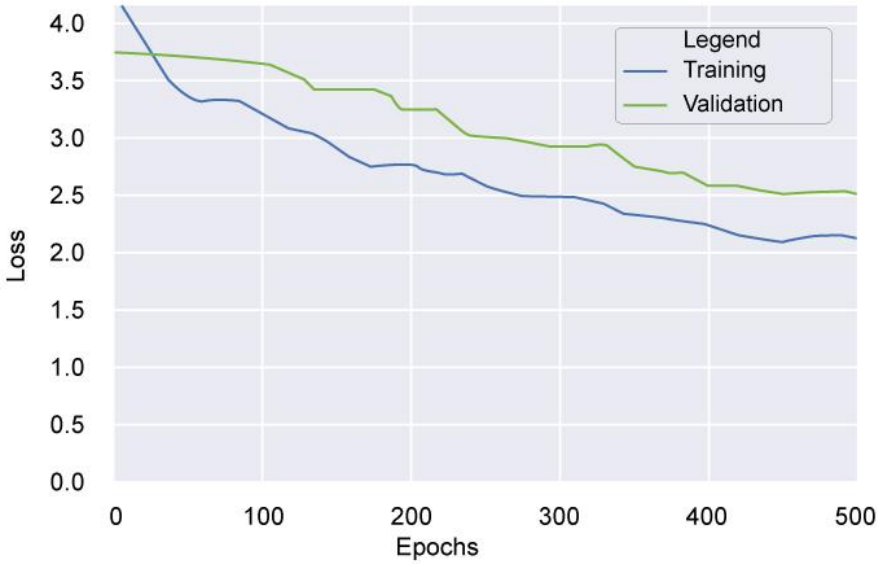


Figure 25: Behavior of training and validation loss when the model is underfitted.

To cope with this problem it has been necessary to tune the learning rate. Figure 25 shows how a low lr leads to the opposite problem: underfitting, i.e. the model is unable to capture the relationship between the input and output. As it can be seen, choosing the learning rate is challenging: a too-small value may result in a long training process that could get stuck, whereas a too-large value may result in learning a sub-optimal set of weights too fast or an unstable training process.

After some parameters tuning a satisfactory outcome about the configuration of the dynamic model is achieved as shown in Figure 26:

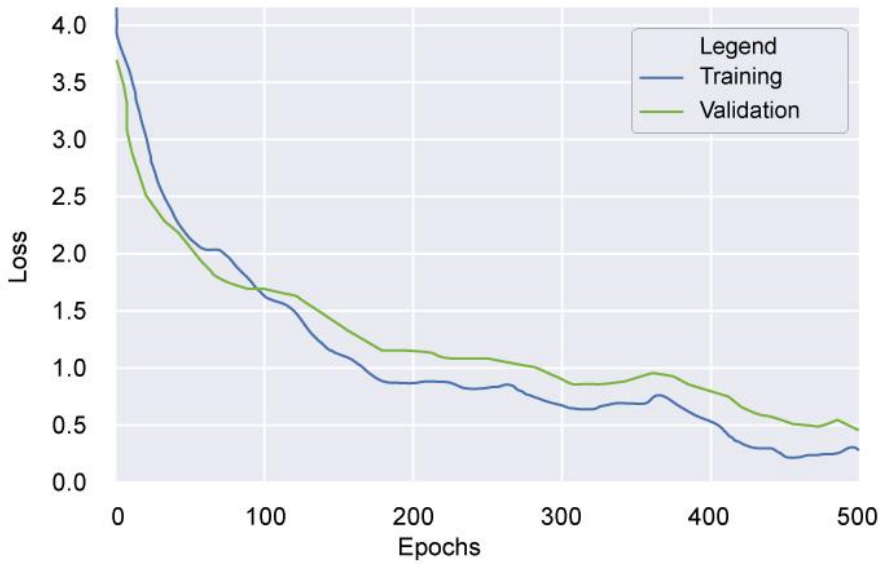


Figure 26: Loss function trend after implementing data replay buffer.

As can be seen, not only the trend of the training loss and validation loss are smoother compared to the one without replay buffer, but they achieve also better performance. Once the final configuration is chosen, a set of simulations is performed during the training to provide an unbiased model performance in terms of episodic accumulated reward, episodic accumulated cost, and sample efficiency. The algorithm is evaluated for 50 episodes, each consisting of 1000 steps. Since the environment is naturally stochastic, the algorithm does not have any previous information about it. In Figures 27 and 28 the episodic accumulated reward and the episodic accumulated cost for the MPC - RS algorithm. A dashed line is used to represent the value at convergence for a model-free approach, since it requires several orders of magnitude more interaction steps to obtain satisfactory performance.

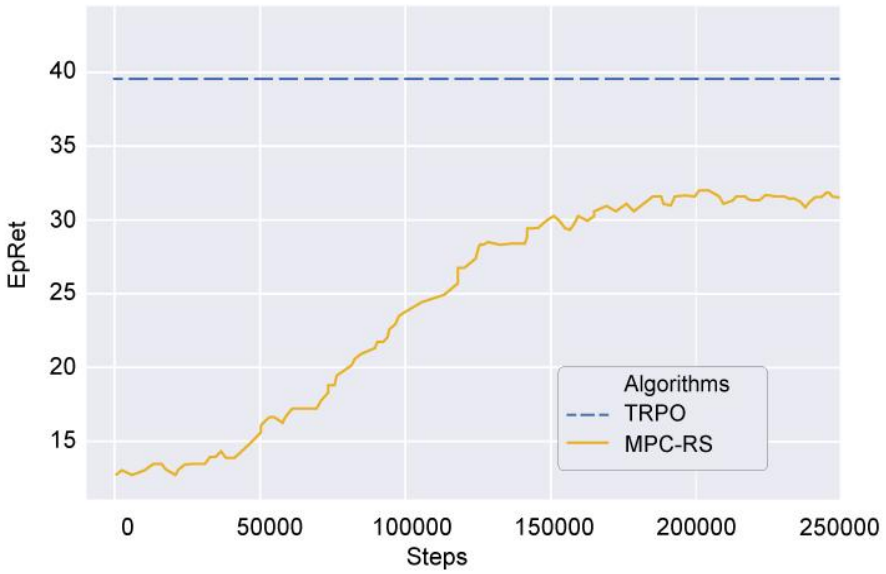


Figure 27: MPC-RS episode accumulated return.

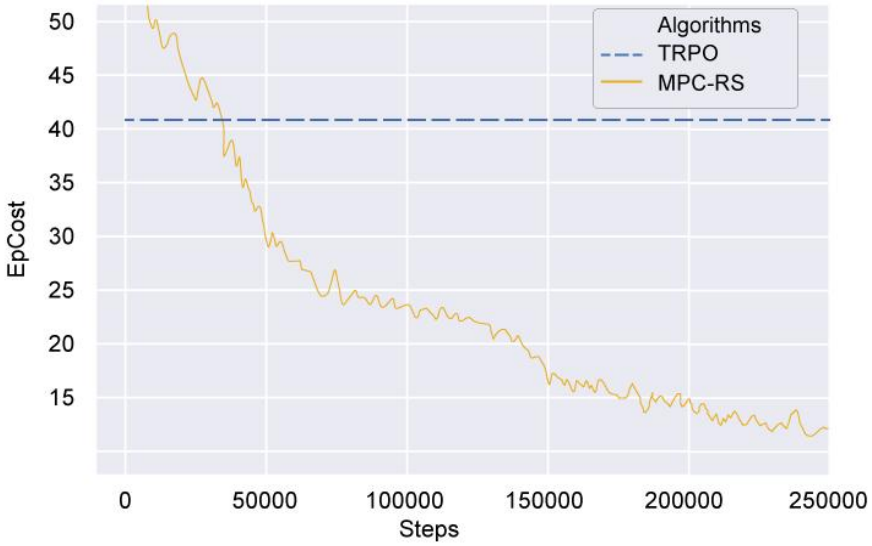


Figure 28: MPC-RS episodic accumulated cost.

From the figures (21, 22, 27, 28), it is evident that the MPC-RS algorithm learns the constraint function very quickly to avoid unsafe behaviors and gets fewer costs, though its reward is lower than the TRPO algorithm. This is reasonable because the best policy to maximize the task reward is to ignore the constraints and let the

Robot go straight towards the goal. It should be noticed that TRPO does not take into account the safety constraints satisfaction but it focuses only to obtain the

highest possible task reward. It is also evident that the model-based MPC-RS algorithm is more sample efficient compared to the model-free method since it reaches convergence, more or less, ten times before.

Finally, further consideration can be done. Firstly, the usage of a replay data buffer significantly enhances performance, revealing the benefits of improving the learned model with additional data. However, the drawback is that the training requires much more time to be done, as well as the computational cost is very high. Secondly, there could be cases in which the robot does not follow “linear” trajectories, but its path to reach the goal is discontinuous. Therefore, to overcome this issue specific path constraints can be added.

CHAPTER 4

4.1 Conclusion

This work introduces a model-based MPC algorithm without any assumption on both the system dynamics and the constraint function, which are learned from the collected data, within the Constrained Markovian Decision Processes framework. The algorithm is able to learn a neural network transition function for a simulated locomotion task using a reduced number of samples. We used a simple Random Shooting method in combination with MPC in order to optimize the choice of the right action. The analysis is performed in the Safety Gym environment, the first benchmark of a high dimensional continuous environment, and especially in the car goal task in which the robot has to reach a target avoiding an unsafe area. A TRPO algorithm is used as a baseline for the comparison and we observed that the MPC algorithm is able to maintain a near optimal task performance while achieving better constraint satisfaction compared to the TRPO algorithm.

4.2 Future Work

We can consider many good prospects for future work. The simplest future line to adopt could be to improve the cost model to provide a longer horizon risk prediction. Additionally, as opposed to a single Deep Neural Network it could be used a neural network ensemble model to reduce the source of uncertainty due to the lack of data. Generally speaking, an ensemble can be considered a learning technique where many models with generalization ability are joined to solve a problem since they tend to perform better than single ones. Another aspect of future work includes improving the MPC controller. In this work, the Random Shooting optimization method is chosen both for its simplicity and for its low computational resource consumption but this may not be the best solution, especially when deployed on real robotic systems or in high-dimensional actions space environments that could require a large number of actions to be sampled. Lastly, a Reinforcement Learning

(RL) approach can be considered. RL algorithms are very powerful, and for this reason, they could be used in place of as well as in combination with the MPC algorithm to improve performance. In particular, in the latter case, the algorithm of this thesis could be used to initialize a model-free RL algorithm to combine the sample efficiency of the model-based approach with the high performance of the model-free approach. A further expectation is that along with the measure of the performance of a system, also the “safety” will be measured as well in order to become a standard for AI developers across the AI sector.

References

[1] *Artificial Intelligence, Automation, and the Economy* - Jason Furman, John P. Holdren, Cecilia Muñoz, Megan Smith, Jeffrey Zients

[2] *Safe and Fast Tracking on a Robot Manipulator: Robust MPC and Neural Neural Network Control* - Julian Nubert, Johannes Kohler, Vincent Berenz, Frank Allgower, and Sebastian Trimpe

[3] *A survey of numerical methods for optimal control* - Anil V. Rao

[4] *Constrained Model-Based Reinforcement learning with robust Cross-Entropy Method* - Zuxin Liu , Hongyi Zhou, Baiming Chen, Sicheng Zhong, Ding Zhao

[5] *Safe and Near-Optimal policy learning for Model Predictive Control using Primal-Dual Neural Networks* - Xiaojing Zhang, Monimoy Bujarbaruah, and Francesco Borrelli

[6] *Exploring Model-Based Planning with Policy Networks* - Tingwu Wang, Jimmy Ba

[7] *A Dual Approach to Constrained Markov Decision Processes with Entropy Regularization* - Donghao Ying, Yuhao Ding, Javad Lavaei

[8] *An introduction to Reinforcement Learning* - https://deepanshut041.github.io/Reinforcement-Learning/notes/00_Introduction_to_rl/"

[9] *Learning- based Model Predictive Control for Markov Decision Processes* - Rudy R. Negenborn, Bart De Schutter, Marco A. Wiering, Hans Hellendoorn

[10] *Benchmarking Safe Exploration in Deep Reinforcement Learning* - Alex Ray, Joshua Achiam, Dario Amodei

[11] *Online Planning Based Reinforcement Learning for Robotics Manipulation* -

[12] *Lecture Notes in Computer Science: Safe Exploration Techniques for Reinforcement Learning - An Overview* - Martin Pecka, and Tomas Svoboda

[13] *Data Efficient Reinforcement Learning for Legged Robots* - Yuxiang Yang, Ken Caluwaerts, Atil Iscen, Tingnan Zhang, Jie Tan, Vikas Sindhwani

[14] *Model Predictive Control fundamentals* - P.E. Orukpe

[15] *Neural Network Dynamic for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning* - Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, Sergey Levine -University of California, Berkeley

[16] *Model-Based Predictive Control: A Practical Approach* - Robert H. Bishop

[17] *Constrained model predictive control: Stability and optimality* - D. Q. Mayne, J. B. Rawlings, C. V. Rao, P. O. M. Scokaert

[18] *An integrated system for real-time Model Predictive Control of humanoid robots* - Tom Erez, Kendall Lowrey, Yuval Tassa, Vikash Kumar, Svetoslav Kolev and Emanuel Todorov

[19] *Model-Predictive Control via Cross-Entropy and Gradient-Based Optimization* - Homanga Bharadhwaj, Kevin (Cheng) Xie, and Florian Shkurti

[20] *Model Predictive Control* - Edward S. Meadows, James B. Rawlings

[21] *Adding Terrain Height to Improve Model Learning for Path Tracking on Uneven Terrain by a Four Wheel Robot* - Rohit Sonker and Ashish Dutta

[22] *Nonlinear Model Predictive Control: Theory and Algorithms* - Lars Grune, Jurgen Pannek

[23] *Understanding Model-Based Reinforcement Learning and its Application in Safe Reinforcement Learning* - Hu, Dingcheng

[24] *Survey of Model-Based ReinforcementLearning: Applications on Robotics* - Athanasios S. Polydoros and Lazaros Nalpantidis

[25] *Reinforcement Learning Algorithms with Python* - Andrea Lonza

[26] *RL — Model-based Reinforcement Learning* - Jonathan Hui

[27] *A Survey on Policy Search for Robotics* - Marc Peter Deisenroth, Gerhard Neumann and Jan Peters

[28] *Continuous Deep Q-Learning with Model-based Acceleration* - Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, Sergey Levine

[29] *When to Trust Your Model: Model-Based Policy Optimization* - Michael Janner, Justin Fu, Marvin Zhang, Sergey Levine

[30] *Hands-On Mathematics for Deep Learning: Build a solid mathematical foundation for training efficient deep neural networks* - Jay Dawani

[31] *Python Deep Learning: exploring deep learning techniques and neural network architectures with PyTorch, Keras, and TensorFlow* - Ivan Vasilev, Daniel Slater, Gianmario Spacagna, Peter Roelants, Valentino Zocca

[32] *Learn Keras for Deep Neural Networks: A Fast-Track Approach to Modern Deep Learning with Python* - Jojo Moolayil

[33] *Adam: A Method For Stochastic Optimization* - Diederik P. Kingma, Jimmy Lei Ba

[34] *PyTorch: An Imperative Style, High-Performance Deep Learning Library* - Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, Soumith Chintala

[35] *Trust Region Policy Optimization* - John Schulman, Sergey Levine, Philipp

Moritz, Michael Jordan, Pieter Abbeel