UNIVERSITÀ DI PARMA

UNIVERSITA' DEGLI STUDI DI PARMA

DOTTORATO DI RICERCA IN
INGEGNERIA INDUSTRIALE

CICLO 33°

Leveraging Non-Smooth Multibody Dynamics and Deep Reinforcement Learning to infer control policies for autonomous robots and vehicles

Coordinatore:
Chiar.mo Prof. Gianni Royer

Tutore:
Chiar.mo Prof. Alessandro Tasora

Dottorando: Simone Benatti

Anni Accademici 2017/2018 – 2019/2020

Simone Benatti

Leveraging Non-Smooth Multibody Dynamics and Deep Reinforcement Learning to infer control policies for autonomous robots and vehicles

November 3, 2020

Contents

Acronyms

Use the template *acronym.tex* together with the Springer document class SVMono (monograph-type books) or SVMult (edited books) to style your list(s) of abbreviations or symbols in the Springer layout.

Lists of abbreviations, symbols and the like are easily formatted with the help of the Springer-enhanced `description` environment.

MDP	Markov Decision Process
POMDP	Partially Observable Markov Decision Process
MRP	Markov Reinforcement Process
RL	Reinforcement Learning
DL	Deep Learning
DRL	Deep Reinforcement Learning
PG	Policy Gradient
DNN	Deep Neural Network
NN	Neural Network
CNN	Convolutional Neural Network
FCL	Full Connected Layer
HL	Hidden Layer
PPO	Priximal Policy Optimization
DDPG	Deep Deterministic Policy Gradient
DQN	Deep Q-Network
w.r.t	with respect to
VF	Value Function
AF	Advantage Function
DOF	Degree(s) Of Freedom
MBD	Multi Body Dynamics

Chapter 1
Introduction

The last few years have seen the rise of Deep Reinforcement Learning Techniques. After its introduction [30], DRL has proved capable of solving complex tasks in various areas, such as video games [10] and robotic manipulation [31].

The reason of its success are not limited to its control capabilities. DRL is model-free, since it only communicates with the environments through state-action-reward tuples signals. This makes DRL completely environmen-agnostic, to the point that the same algorithm can be used in completely different scenarios. Moreover, DRL can deal with large, raw inputs, such as RGB pixels.

The need for large amount of data, in the from of RL signal tuples, has risen the Machine Learning community a great interest in physics simulation (and MBD in particular) to provide virtual environments in which DRL agents can be trained without risks and with limited cost.

The vast majority of DRL algorithms are written in Python, since the most used Deep Learning frameworks provide Python API. This has made Python API a strong requirement for physics simulation libraries to be used in the context of DRL. The reason of Python success in computational science is its ease of use (being an interpreted language) combined with the capability of being interfaced with fast, compiled libraries, thus overcoming the intrinsic performance limitations of interpreted language.

These considerations made evident the significance of the development of comprehensive Python bindings for our simulation library project Chrono, together with a set of several virtual environments in order to demonstrate its capabilities in various applications, such as robotics and autonomous vehicles.

In addition, we considered a propriety to study and develop DRL algorithms, to both show the capabilities of our virtual environments and to investigate the synergies between simulation and DRL.

Moreover, we wanted these tools to be as accessible as possible, such that researcher from companies or other universities can take advantage of our open source framework. .

Part I
State of the art in Deep Reinforcement Learning and Multi Body Simulation

In the following chapters some basic concepts of RL are introduced, and we present strategies and algorithms suitable for continuous control tasks. In the 3rd chapter, we introduce numerical techniques to simulate multi body system with contacts.

Chapter 2
Reinforcement Learning

The following chapter is an introduction to Reinforcement Learning, a Machine Learning technique used to infer model-free control policies from agent-environment interaction signals.

2.1 Reinforcement Learning Basics

2.1.1 Markov Decision Process

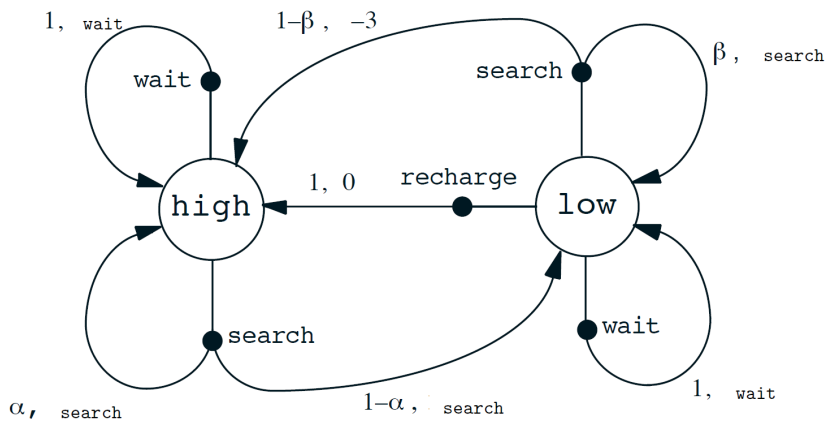


Fig. 2.1: Markov Decision Process transitions example [43]

The Markov Decision Process [43] is a stochastic discrete-time control process where an *agent* interacts with an *environment* by taking an *action*, the influences the

environment *transition* from a *state* to another.The MDP is a memory-less process, in other words the future is independent of the past given the present. A control process can be a MDP if it respect the Markov property, which states that given the current state and action, the next state does not depend on past states. The *transition probability* $p(s'|s_t, a_t) = Pr(S_{t+1} = s'|A_t = a_t, S_T = s_t)$ in a MDP does not depend on $A_0 ... A_{t-1}$ and $S_0 ... S_{t-1}$. In a MDP the information coming from the state is sufficient to solve the task.

POMDP

Let us consider a maze: if someone tries to escape from it, he has no way to know the exact solution but, if the maze is not too complicated, he can somehow get out, obviously not by taking the shortest path but finding a good exploration strategy instead. If the state (also called observation) lacks useful information to solve the task we call it *Partially Observable Markov Decision Process* (POMDP). Please note that a task can respect the Markov Property and still be a POMDP, such as the maze: the next position depends only on the current position and next step direction, but the current observation does not tell anything about the best way out.

MRP

The reward $r_t \in \mathbb{R}$ is a signal that the agent receives at each iteration and it is related to the agent performance. The sum of reward is a measure of how well the task has been done. Let us consider the figure 2.1: the trash-collector robot has 2 possible states (high and low) and take 3 possible actions (search, recharge, wait). In the graph, each action from a state has one or more possible outcomes, for example: doing search in high state can lead to high again with probability α or to low state with probability $1 - \alpha$. Please note that the search and wait reward value are to be assigned while the recharge reward is 0 and the manual rescue (it happens with probability $1 - \beta$ doing search in low state) reward is -3. It is evident that

- The choice of the action in a state determines the outcome in terms of sum of rewards
- The choice of the reward assigned in a given situation affects the behavior associated with the maximum reward

2.1.2 Reinforcement Learning Motivation and Elements

We previously mentioned some fundamental concepts of MDP (and RL), such as Environment, Agent, Action, State etc which are more deeply explained here.

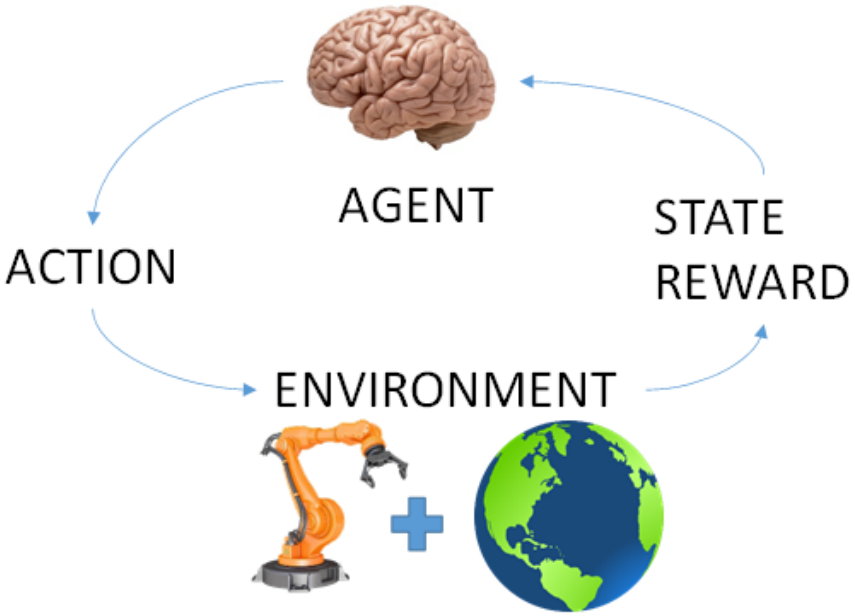


Fig. 2.2: The Agent-Environment interaction in the RL process.

The Agent and the Environment

What we call agent is the decision-maker, while whatever the agents interacts with is part of the environment. The border between the agent and the environment is logical, not physical: everything the control has no direct control on is part of the environment. Let us consider a torque-controlled robot: while the torques applied by the motors are part the agent, the robot state (positions and velocities) is definitely part of the environment. This is the reason why in figure 2.2, the robot and its surrounding (the world) are both represented in the environment.

The State

The state, also called *Observation* is whatever information available to the agent. This information should respect the Markov property. The the state signal can come in any shape: vectors of real numbers, RGB images, depth buffers or any combination of these.

The Reward

The goal of the agent is formalized as a reward signal passing given by the environment to the agent. The agent's objective is the maximization of the sum of collected rewards. The so called *Return* is the sum of the rewards collected or, more often, the sum of discounted reward. Discounting the reward means multiplying the future rewards by an exponentially decaying *discount factor* $\gamma. \in (0,1)$ as in the second return term in 2.1.

$$G_{t1} = \sum_{t=0}^{\infty} r_t \qquad G_{t2} = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.1}$$

The return is often referred to as objective and indicated with J

2.1.2.1 The Policy

We know that the agent must maximize the sum of collected reward by doing the only thing it can do, choosing the right action, based on the only information available to it, the state. The policy is exactly this: the strategy prescribing the action to take according to the state, generally it is a mapping from states to probabilities of selecting each possible action.

Reinforcement Learning makes use of Machine Learning techniques to find a policy, while *Deep Reinforcement Learning* (DRL) means that the ML technique used involves Deep Learning. The reason for using a ML technique to infer a policy is evident when considering the so-called *curse of dimensionality*: in the task represented in figure 2.1 there are only 2 states and 3 possible action in each state, thus is easy to sample the outcome of each state-action pair and find the optimal policy. In most real-world situation the number of state and action dimensions cause an exponential explosion of the state-action possible pairs, making the complete exploration of the environment infeasible, thus making ML approaches the only viable option.

Continuous vs Discrete action spaces

Let us consider a RL agent trying to win an arcade game and another one controlling a n-DOF robotic arm by imposing the actuator torques: the first at each timestep will have to pick an action from a finite set of possible actions, while the latter action has to be chosen from the cartesian product of n subsets of $\mathbb{R}$. Control tasks whose set of possible actions is finite (usually a cartesian product of boolean and/or integer intervals) such as the arcade game mentioned before are called *Discrete Action Spaces*, while when the set of actions is continuous we call it a *Continuous Action Space* task. It is evident that given the difference between these problem families, algorithm developed for one of these classes will perform poorly (or often they cannot be applied) on the other one. These concepts will be investigated further in this section.

2.1.3 Strengths and Limitations of RL

Generality

The RL approach can, in principle, be applied to any control task. From robotics to videogames the presence of an agent interacting with an environment whose actions have to accomplish a task based on the input received is common, thus many control problem can be formalized as MDP, so RL can be applied to solve them. Applications of RL have shown impressive results in table games, retro arcade games [30] and modern strategy video games [10], autonomous driving and all field of robotics: pick and place [53], walking robots [44] and manipulation [31].

TODO: cite alphaGO, Minh, Alpha star and dota5, AV paper, stanford and minitaur and learning dexterity

Model-Free

In the MDP scheme the environment is a black box, and the vast majority of RL algorithms don't make any further assumption about the knowledge of the environment transition. This means that

- ML algorithms can be trained directly in the real world
- There is no need for analytical models of the environment
- Simulated environments are needed only for the sampling

The advantage of the latter point is not to be underestimated under the software implementation point of view, since it means that the ML algorithm does not need any knowledge of the environment and vice versa, thus the algorithm and the simulated environment are easy to interface since they only communicate the through the state, action reward signal.

Exploration

A RL agent explores the state space by sampling through interacting with the environment, and the way the agent interacts with the environment is determined by the policy. Even though all RL algorithms provide some kind of random exploration, bad policies lead to bad exploration, meaning that the agent is never getting high rewards, thus not being able to train towards a good behavior. This policy-sampling mutual interaction arises 2 issues:

- Starting from an untrained (random) policy, the agent might not be able to improve.
- An almost trained policy can diverge after a bad update.

Reward Sparsity

The easiest way for the agent to learn a policy is through dense reward signals, such that the ML algorithm can evaluate the goodness of a batch of actions by looking at the rewards collected in the adjacent timesteps, but this is not always possible. Let us consider a pick-and-place task where a 6-DOF robotic arm hast to pick the right object and put it on a shelf. We could define the reward to be 1 when the right object is on the shelf, and 0 otherwise. In this case the RL agent should train the policy while always getting 0 reward, since it is impossible for the agent to solve the task by chance, and the RL algorithm can not be trained from a constantly 0 reward. When dealing with this kind of problems the possible solution are engineering the reward to obtain a denser reward function, or use algorithm able to deal with *Sparse Reward*.

The reward engineering is a tricky solution, since the reward is the way we tell the agent what we want it to accomplish, and changing it can lead to unwanted behaviors; while algorithm suitable for sparse reward environments might not be able to solve the task anyway.

2.2 Value Functions

2.2.1 Introductory Statistics Concepts

Here some probability and statistics basic notions are briefly introduced.

2.2.1.1 Random Variables

We define ***Random Variable*** a variable whose value is random, while its ***probability distribution*** represents its likeliness to take on each of its possible states. Random variables can be *Discrete* or *Continuous*: definitions and formulas are different for the two categories, although being one the transposition of the other on a dense dominion (or vice versa).

The *Probability Mass Function* (PMF) and *Probability Density Function* (PDF) are an example of this: $P(x)$ is the probability of the variable being x if the variable is discrete, while it is the probability of the variable being in $[x-\delta x; x+\delta x]$ if the variable is continuous. We call a PMF to be normalized when the sum over all possible values is 1, while a PDF is normalized when its integral over the whole dominion is 1.

Expected Value and Variance

The **expected value** (also called expectation) of a function $f(x)$ w.r.t a probability distribution $P(x)$ is the mean value that f takes on if x is drawn from P. For discrete and continuous random variables respectively, it is computed as:

$$\mathbb{E}_{x\sim P}(f(x)) = \sum_x P(x)f(x) \tag{2.2}$$

$$\mathbb{E}_{x\sim P}(f(x)) = \int_P (x)f(x)dx \tag{2.3}$$

Many times the expectation is written as $\mathbb{E}[f]$ and the distribution drawing ($x \sim P$) is implied. The **Variance** indicates how much a random variable varies:

$$Var(f(x)) = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])\right] \tag{2.4}$$

2.2.1.2 Conditional Probability

In many situations, random events (formalized as random variable) are somehow correlated, and knowing whether one of these happened or not changes the likelihood of the other to happen. We call the conditional probability that $y = \hat{y}$ given $x = \hat{x}$ **Conditional Probability** and it is denoted as $P(y = \hat{y}|x = \hat{x})$. Conditional Probability is evaluated as follows:

$$P(y = \hat{y}|x = \hat{x}) = \frac{P(y = \hat{y}, x = \hat{x})}{P(x = \hat{x})} \tag{2.5}$$

Conditional probability is often written $P(y|x)$, denoting probability of a generic value of y knowing the value of x.

Bayes' Rule

The Bayes' Rule states that:

$$P(x|y) = \frac{P(x)P(y|x)}{P(y)} \tag{2.6}$$

2.2.2 Value Functions

Given a state, or an action taken in a state, it is common to wonder how being in a state, or taking a particular action in a particular state is ***good***, but first we have to define what *good* means in RL context. Since, recalling equation 2.1, we want to

maximize the sum of discounted reward, a state (or state-action pair) is as good as the sum of discounted rewards the we *expect* to get from there. Since the way the agent interacts with the environment is governed by the policy, the value function is always referred to a particular policy π and. Since both the policy (and often the environment as well) is stochastic, the value function is an the *expected value* of the future reward.

In the following paragraphs there is no mention about how those function are evaluated, since it is not normally possible directly compute them. As will be shown in chapter 4, many algorithms rely on Deep Learning techniques to estimate one of these functions.

State Value Function

We define the *State Value Function* as:

$$V^{\pi}(s_t) \quad := \quad \mathbb{E}[G_t \mid s_t] \quad = \quad \mathbb{E}_{\substack{s_{t+1:\infty}; \\ a_{t:\infty}}} \left[\sum_{l=0}^{\infty} \gamma^l r_{t+l} \right] \tag{2.7}$$

Where:

- G_t is the Return at timestep t
- s_t is the State at timestep t
- r_{t+l} is the Reward at timestep $t+l$
- γ is the discount factor

The expectation subscript in the last part of the equation is an alternative formulation for $|s_t$, where $a_{t:\infty}s_{t+1:\infty}$ means that the expectation involves the state from s_{t+1} and the action starting from a_t.

The state value function definition from equation 2.7 means that the value function $V^{\pi}(s_t)$ is the expected outcome in state s_t while following the policy π.

Finite and Infinite horizon

In 2.7 we considered an infinite-horizon MDP (the sum ends at ∞), while in practice every agent-environment interaction reaches an end sooner or later: the agent might just achieve the desired goal, or meet a severe failure (like a car crashing or a walker falling on the ground) or exceed the maximum time allowed to solve the task. These case are called *finite-horizon*.

In finite-horizon the last timestep of an interaction T is called *terminal* and the agent-environment interaction between the initial and terminal step in referred to as *Episode*.

Many formulas are given for the infinite-horizon case for generality but finite-horizon is just a sub-case in which the sum interrupts at T.

The Bellman Equation

Value functions satisfy recursive relationships, such as the Bellmann Equation for the value function V^π between the value of s and the value of its expected successor states .

$$\begin{aligned} V^\pi(s_t) &:= \mathbb{E}[G_t \mid s_t] \\ &= \mathbb{E}\left[\sum_{l=0}^{\infty} \gamma^l r_{t+l} \mid s_t\right] \\ &= r_t + \mathbb{E}\left[\sum_{l=1}^{\infty} \gamma^l r_{t+l+1} \mid s_t\right] \\ &= r_t + \sum_a \pi(a \mid s_t) \sum_{s_{t+1}} p(s_{t+1} \mid a_t s_t) \mathbb{E}\left[\sum_{l=1}^{\infty} \gamma^l r_{t+l+1} \mid s_{t+1}\right] \\ &= r_t + \sum_a \pi(a \mid s_t) \sum_{s_{t+1}} p(s_{t+1} \mid a_t s_t) V^\pi(s_{t+1}) \end{aligned} \tag{2.8}$$

State-Action Value Function

Similarly to the state value function, the state-action value function is an expectation of he outcome of taking a particular action in a given state, following the policy thereafter.

$$Q^\pi(s_t, a_t) := \mathbb{E}[G_t \mid s_t, a_t] = \mathbb{E}_{\substack{s_{t+1:\infty}; \\ a_{t+1:\infty}}}\left[\sum_{l=0}^{\infty} \gamma^l r_{t+l}\right] \tag{2.9}$$

The state-action value function, a.k.a. Q-function, tells directly how good taking an action in a particular state is. In other words, knowing the Q-function of each possible action (this is possible in discrete-action environments) means knowing which action is more likely to lead to the highest return.

Advantage Function

The *Advantage Function* is defined as:

$$A^\pi(s_t, a_t) := Q^\pi(s_t, a_t) - V^\pi(s_t) \tag{2.10}$$

The Advantage function carries a meaningful information, since it directly compares the return from taking the action a_t ins s_t and then following the policy π to directly following the policy in s_t, thus the higher $A^\pi(s_t, a_t)$ is, the better is taking that action compared to following the policy.

Chapter 3
Introduction to Deep Learning

Deep Learning is a Machine Learning technique that revolves around the use of Deep Neural Networks. Here we introduce the concepts of Neural Networks, Layers, how these functions are fitted and the most common types of layers and outputs.

3.1 Neural Networks

A *feedforward neural network* or *multi-layer perceptron* defines a mapping $\mathbf{y} = f(\mathbf{x}, \theta)$. A Neural Network (NN) is feedforward if there's no connection between its outputs and inputs, while if such connections exist it is said to be a *Recurrent Neural Network* (RNN). For sake of simplicity, here we will focus only on feedforward networks. NN are called "Networks" since they are represented by composing together many functions, and "Neural" since their inspiration comes from neuroscience.

The operations are usually organized into *layers*, and the number of layer gives the *depth* of the Neural Network, and the dimensionality of these hidden layers determines the *width* of the model. The first and last layers are called *Input Layer* and *Output Layer* respectively, while the layers between these are called *Hidden Layers*. The objective of NN training is tuning the parameters θ in order to match an objective $f^*(\mathbf{x})$.

3.1.0.1 Single Layer Perceptron

The basic unit of a NN is the *Single Layer Perceptron* or *Neuron*, represented in figure 3.1. The output value of this elementary unit is given by a function called *Activation Function* (see 3.1.1.3) whose argument is the dot product of the input vector $\mathbf{x}$ and the weight vector $\mathbf{w}$.

$$y = f(\mathbf{x} \cdot \mathbf{w}) \tag{3.1}$$

It can be easily noticed that the activation input maps linearly to the input, thus the fitting capability of the single neuron is very limited. This being said, it is also easy to find the derivative of the error with respect to the set of weights. Given a target t and an error function, such as $E = \frac{1}{2}(t-y)^2$ we can apply the chain rule of differentiation as follows:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial x \cdot w_i}\frac{\partial x \cdot w}{\partial w_i} \tag{3.2}$$

$$= (t-y)f' x_i \tag{3.3}$$

If the function f is differentiable, it is always possible to evaluate the gradient of the error with respect to each weight. This property comes in handy when fitting the weight with Gradient Descent (see 3.1.1). With due modifications, it is possible to extend this to complex NN architecture, as will be showed in 3.2.0.1.

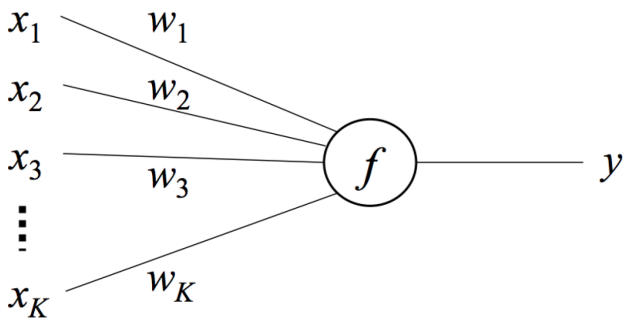


Fig. 3.1: A Neuron, or single layer perceptron.

3.1.0.2 MLP

Neural Networks are meant to overcome the limits of SLP in terms of function approximation, and (as the name suggests) consist of multiple layers of neurons. Each layer consists of several neurons and it is the input of the following layer, and every layer between the input and the output is called *hidden layer*. In terms of function approximation the universal approximation theorem [23] states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathbb{R}^n$ (under assumptions on the activation function). This being said, even if a NN is capable of fitting a function there's no guarantee that the algorithm will successfully fit its weight, since there is no universally superior machine learning algorithm (the *no free lunch theorem* [51]).

Moreover, even if the universal approximation theorem states that one hidden layer is sufficient, the width of the single hidden layer could be so large for the ML algorithm to manage. For this reason it is usually preferred having multiple hidden layers to contain the NN width [16]. The technique shown to fit the weights in the SLP can be extended, with due modification, to any NN.

Dense and sparse connections

In SLP the input is fully connected to the input, otherwise the unconnected inputs would be ignored, but this is not always the case in more complex architectures, in which a neuron could be connected only to some of the neurons of the following layer, and we call it a *sparse connectivity*, of which Convolutional Neural Network (see 3.3) are the most widely known example. If each neuron is connected to every neuron of the following layer the layer instead, it is said to be *Fully Connected* (FCL). Let us consider a FCL as represented in figure 3.2, where the layer k of n neurons is fully connected to the layer l of m neurons (thus there are $n \cdot m$ weights). The value in the neuron j in the layer l is

$$x_j = g\left(\sum_{i=1}^{n} w_{ij} x_i + b_j\right) \tag{3.4}$$

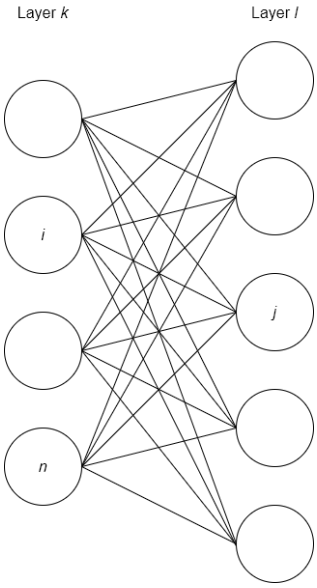


Fig. 3.2: Representation of a Fully Connected Layer.

Where w_{ij} is the weight connecting the i-th neuron of the k-th layer with the j-th neuron of the l-th layer, and b_j is the bias summed to the j-th neuron. A general expression for an hidden layer can be:

$$\mathbf{h}^{(i)} = g^{(1)}\left(\mathbf{W}^{(i)^\top}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)}\right) \tag{3.5}$$

Where h^i is the i-th hidden layer $\mathbf{W}^{(i)}$ and $\mathbf{b}^{(i)}$ are respectively the matrix of weights and the vector of biases. The 3.5 can generalize to any kind of connectivity: if the k-th neuron of the i-th layer is not connected to the l-th neuron of the i-1 layer, then $W^{(i)}_{l,k} = 0$.

3.1.1 Gradient-based learning

3.1.1.1 Cost Function

Neural Networks nonlinearity causes loss functions to become non-convex, thus they are trained using iterative, gradient-based optimizers that aim to reduce the cost function value. Stochastic gradient descent applied to non-convex loss functions is not guaranteed to converge,and is highly parameter sensitive. The optimizers used to train NN described in 3.2 are always based on gradient descend. The specific algorithms are improvements of gradient descent algorithm.

3.1.1.2 Output Units

The NN output choice is related to the cost function and, especially in Deep Reinforcement Learning, to task to accomplish. One of the simplest (and yet often used) output unit are linear units, typically used to produce the mean of a Gaussian distribution.

Some tasks, such as classification problems require predicting the value of a binary variable y (Bernoulli distribution), thus limiting the output value in the $[0,1]$ interval. Clipped linear unit would not work with gradient methods. In this case Sigmoid units are used, since continuous and differentiable everywhere and their output is between 0 and 1. To classify over multiple classes (Multinoulli distribution) Softmax units are used instead. Further details about these outputs unit are beyond the scope of this work, which is focused on Reinforcement Learning for control, rather than classification problems.

In the control applications of Deep Learning *tanh* output function are commonly used, for the following reasons:

- *tanh* is continuous and differentiable over all $\mathbb{R}$
- Its output is between -1 and 1, and multiplied by a gain it can easily model a control or actuator.

Similarly to linear units, *tanh* output units can be used to produce the mean of a Gaussian distribution from which sample the value.

3.1.1.3 Hidden Units Activation Function

In section 3.1.0.1 we mentioned *Activation Functions*. The apply an element-wise non linear function f to the affine transformation $\mathbf{W}^\top \mathbf{h} + \mathbf{b}$ and they are responsible for the layer non linearity. An activation function should be:

- Non linear, if we want the universal approximation theorem to hold
- Monotonic [52]
- Limited, to avoid divergence in gradient optimization
- Differentiable (actually some activation function, such as ReLU, are not differentiable in one point)

Rectified Linear Units

The rectified linear unit, called ReLu is the most widely used hidden unit and they use activation function $f(z) = max(0,z)$.

A rectified linear unit is similar to a a linear unit, besides that its output is zero across half the domain. Thus the derivative through a ReLU remains large if the unit is active, so the gradients are large and consistent, and the derivative of the rectifying operation is 1 everywhere that the unit is active

There are variants to ReLU introduced to have a non-zero gradient when the input is negative. The simplest and most known is the Leaky ReLU, $f(z) = max(0,z)$ where k is a very small constant. Its gradient is no longer 0, but k, when $z < 0$.

Logistic Sigmoid and Hyperbolic Tangent

Other popular activation functions for hidden units are the Logistic Function

$$f(z) = \frac{1}{1+e^{-z}} \tag{3.6}$$

And the hyperbolic tangent, the well-known

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{3.7}$$

Both these function are monotonic, non linear and differentiable everywhere. Their major issue is that $\nabla_z f \to 0$ when $z \to \pm\infty$

3.2 Parameter Update

3.2.0.1 Back-Propagation

In order to apply gradient-based learning methods to a NN the gradient of the objective function with respect to the weights are needed, just as for the SLP in 3.1.0.1.

Applying chain rule to tensors

While large multi-dimensional tensors may look menacing, evaluating their gradient is not conceptually complicated and not much different from what was shown for the SLP. The goal of Back-Propagation is evaluating the derivative of a scalar (typically the loss function) with respect to the weights. Let us consider $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^m$, and let f and g both be functions mapping from $\mathbb{R}^m$ to $\mathbb{R}$ and from $\mathbb{R}^n$ to $\mathbb{R}^m$. If, $y = g(x)$ and $z = f(y)$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial y_j}{\partial x_i} \frac{\partial z}{\partial y_j} \tag{3.8}$$

Written in vector notation it becomes

$$\nabla_{\mathbf{x}} z = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \mathbf{y} \nabla_{\mathbf{y}} z \tag{3.9}$$

The operation to perform boils down to multiplying the Jacobian matrix by the gradient, but in order to extend this to DNN it has to be extended to tensors. In order to do so, let us consider the tensors $\mathbf{X}$ and $\mathbf{Y}$, and refer to their elements using i and j indexes respectively. Here indexes are no longer integers, but a tuple of integers, whose length depends on the number of dimension of the tensors the refer to. For example, is $\mathbf{X}$ is a 3D vector, i will be a 3-elements tuple. For this purpose reshaping an n-dimensional tensor into a vector $\in \mathbb{R}^{d_1 \times d_2 \ldots d_n}$ can be conceptually valid. Thus, we can write:

$$\nabla_{\mathbf{X}} z = \nabla_{\mathbf{X}} \mathbf{Y}^\top \nabla_{\mathbf{Y}} z \tag{3.10}$$

Note that the transpose operation may not be univocally determined, since it is not obvious which indexes are swapped. In this case we are swapping the indexes tuples i and j previously mentioned.

Strategies for Computational Graphs

Dealing with large tensors and their weights (several weights per tensor element) inevitably leads to computational complications, and the CPU and memory usage become prohibitive if the following strategies are not undertaken. In a back-propagation algorithm the same gradient will be used by all its parent node in the computational graph, so strategies such as storing gradient values can strongly

improve performance, avoiding exponential explosion of computational cost. In a backprop implementation for a DL library, the backprop algorithm must generalize to any computational graph, and be applied to symbolic operations. There are different strategies to implement this procedure, know as *symbol-to-number* differentiation (used by PyTorch [33]) and *symbol-to-symbol* differentiation (used by TensorFlow [1]). Further details about backprop implementations in modern DL library would exceed the scope of this work, and a practical example is given instead to clarify the backprop workflow.

In order to operate a backprop we need the objective function J to optimize (here we minimize a loss function L), thus the forward pass is needed to perform backprop.

Algorithm 1: Example of forward pass

Data: l: Network depth
Data: $\mathbf{W}^{(i)}$, $\mathbf{b}^{(i)}, i = \{1...l\}$: weights and biases of each layer
Data: $\mathbf{x}$, $\mathbf{y}$: Input and desired output (target)

$\mathbf{h}^{(\mathbf{0})} = \mathbf{x}$
for k = *1,...,l* **do**
  $\mathbf{a}^{(k)} = \mathbf{W}^{(k)\top}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k)}$
  $\mathbf{h}^{(k)} = f\left(\mathbf{a}^{(k)}\right)$
end
$\hat{\mathbf{y}} = \mathbf{h}^{(l)}$
$J = L(\mathbf{y}, \hat{\mathbf{y}})$

Where

- $\mathbf{a}^{(k)}$ is the *pre-activation* layer of the k-th layer
- f is the activation function. In principle different layers can have different activation function.
- $\mathbf{h}^{(k)}$ is the k-th layer *after* the activation, which is the input of the $k+1$ layer

The backprop can now be performed:

Algorithm 2: Example of backprop in a fully-connected MLP

Data: The forward computation

$\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$
for k = *l, l-1,...,1* **do**
  $\mathbf{g} \leftarrow \nabla_{\mathbf{a}^{(k)}} J = \mathbf{g} \odot f'\left(\mathbf{a}^{(k)}\right)$
  $\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g}$
  $\nabla_{\mathbf{W}^{(k)}} J = \mathbf{g}\,\mathbf{h}^{(k-1)\top}$
  $\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = \mathbf{W}^{(k)\top}\mathbf{g}$
end

Please note that:

- The iteration starts from the last layer and goes backwards until the firs hidden layer; this is the reason why it is called *Back* Propagation

- $\mathbf{g}$ value is assigned multiple times for efficiency reasons to avoid multiple evaluations.
- At line 3 the gradient *before* the update is $\nabla_{\hat{\mathbf{y}}} J$ if $k = l$, and $\nabla_{\mathbf{h}^{(k)}} J$ if $k < l$
- The gradient of the loss function w.r.t. weights and biases (line 4 and 5) is what we need to update the NN parameters using gradient descent.

3.2.1 Optimization

Stochastic Gradient Descent

Optimizing a NN means tuning its parameter in such a way to increase an objective function to maximize (or vice versa). Since the training set is limited, we want to optimize the objective function over a finite number of samples. In other words, we are optimizing a Monte-Carlo estimate of the objective function. This is equivalent to maximizing/minimizing the expectation over a distribution (the training set). Also properties of the objective function are expectations too, such as its gradient. The gradient of the objective function (w.r.t. the NN parameters) is what the back-propagation needs in order to fit the weights and the biases, so given the dataset, there are 2 possible ways to proceed:

1. ***Deterministic gradient descent***: the gradient of the objective function is evaluated over all the samples of the training set
2. ***Stochastic gradient descent***: the gradient of the objective function is evaluated only over a subset of the training set

In both cases we want to compute an *estimate* of the gradient of the objective function, thus the goal is minimizing the error on the estimate while limiting the computational cost. To this extent, we recall that the Standard Error (SE) of the mean when estimating a stochastic quantity from a finite number of samples is [16] :

$$\mathrm{SE} = \frac{\sigma}{\sqrt{n}} \tag{3.11}$$

Where σ is the variance of the distribution and n is the number of samples. This demonstrates that the return from using more samples when estimating the gradient of the objective function is less than linear. On the other hand, parallel architectures (typically these jobs run on GPUs) are underutilized by small batches. In practice, NN are optimized through ***minibatch*** or ***minibatch stochastic*** methods (often referred to simply as *stochastic* methods) that uses several but not all the training set tuples. Typically, the higher derivative order is needed, the larger the batch must be to limit the error on the estimate; for example, a 100 tuples batch can suffice for first order methods, but 10000 batches may be required for second order methods in which the Hessian matrix is required [16]. Another crucial requirement for the training set is to avoid correlation between adjacent samples, since this will lead to biased training. The correlation between adjacent samples is common in several sit-

uations, such as in robotics, where data from simulation are passed in chronological order. Is usually enough to shuffle the samples to prevent samples correlation from biasing the training.

3.2.1.1 Stochastic Gradient Descent

One of the most known and used algorithms, and the first one to be introduced is *Stochastic Gradient Descent*:

Algorithm 3: Stochastic gradient descent

Data: Learning rate ε_k ; Initial NN parameter θ
while *Stop criterion met == false* **do**
  Sample m-elements minibatch with inputs $\{\mathbf{x_1}, \mathbf{x_2}, , , , \mathbf{x_m}\}$ and targets $\{\hat{\mathbf{y_1}}, \hat{\mathbf{y_2}}, , , , \hat{\mathbf{y_m}}\}$
  Gradient estimate: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L((\mathbf{x_i}; \theta)\hat{\mathbf{y}}_\mathbf{i})$
  Weights update:$\theta \leftarrow \theta - \varepsilon \mathbf{g}$
end

Even though more recent and advanced algorithm took place in the last few years these are evolution of the SGD, and they basically refine and tune the operations in 3.

Learning Rate

The term ε in the SGD algorithm is the *Learning Rate* (or LR), and represents the proportion between the gradient of the loss function w.r.t. NN parameters and the parameters. In other words, increasing the LR will enlarge the size of the weight and bias update given the same gradient. LR is a critical parameter in DL: using larger LR may bring to convergence faster and allow to jump across local minima; on the other hand if the LR is too large the parameter update could be disruptive or make the learning unstable. Choosing a LR value is highly empirical, but many optimization algorithms tune it during the training.

3.2.1.2 Momentum

The momentum has been introduced to speed up the learning process, and its name is due to an analogy with mechanics: the gradient updates a velocity quantity v which is then used to update the parameters. This allows the previous updates to affect the parameter update. The algorithm

Algorithm 4: Stochastic gradient descent with Momentum

Data: Learning rate ε_k ; Initial velocity $\mathbf{v}$; Initial NN parameter θ

1 **while** *Stop criterion met* == *false* **do**

2 Sample m-elements minibatch with inputs $\{\mathbf{x_1}, \mathbf{x_2}, , , , \mathbf{x_m}\}$ and targets $\{\hat{\mathbf{y_1}}, \hat{\mathbf{y_2}}, , , , \hat{\mathbf{y_m}}\}$

3 Gradient estimate: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L((\mathbf{x_i}; \theta) \hat{\mathbf{y}}_\mathbf{i})$

4 Velocity update:$\mathbf{v} \leftarrow \alpha \mathbf{v} - \varepsilon \mathbf{g}$

5 Weights update:$\theta \leftarrow \theta + \mathbf{v}$

6 **end**

Note that the multiplication by α of the old velocity at line 4 makes the older updates decay exponentially.

AdaGrad

The AdaGrad (Adaptive Gradient) algorithm adapts the learning rate of each parameter by scaling it inversely proportional to the square root of the sum of all of the history of its gradient.

Algorithm 5: AdaGrad Algorithm

Data: Learning rate ε ; Initial NN parameter θ

Data: Small constant ($\sim 10^{-7}$) δ

1 $\mathbf{r} = 0$

2 **while** *Stop criterion met* == *false* **do**

3 Sample m-elements minibatch with inputs $\{\mathbf{x_1}, \mathbf{x_2}, , , , \mathbf{x_m}\}$ and targets $\{\hat{\mathbf{y_1}}, \hat{\mathbf{y_2}}, , , , \hat{\mathbf{y_m}}\}$

4 Gradient estimate: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L((\mathbf{x_i}; \theta) \hat{\mathbf{y}}_\mathbf{i})$

5 Squared grad accumulation: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$

6 Weights update:$\theta \leftarrow \theta - \frac{\varepsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$

7 **end**

RMS-prop

Similarly to AdaGrad this algorithm adapts LR individually, but uses an exponentially weighted moving average to suppress the influence of older updates.

Algorithm 6: RMS-prop Algorithm

Data: Learning rate ε ; Initial NN parameter θ
Data: Small constant ($\sim 10^{-7}$) δ ; decay rate ρ
$\mathbf{r} = 0$
while *Stop criterion met* == *false* **do**
  Sample m-elements minibatch with inputs $\{\mathbf{x_1}, \mathbf{x_2}, , , , \mathbf{x_m}\}$ and targets $\{\hat{\mathbf{y_1}}, \hat{\mathbf{y_2}}, , , , \hat{\mathbf{y_m}}\}$
  Gradient estimate: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L((\mathbf{x_i}; \theta)\hat{\mathbf{y}}_\mathbf{i})$
  Squared grad accumulation: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1-\rho)\mathbf{g} \odot \mathbf{g}$
  Weights update:$\theta \leftarrow \theta - \frac{\varepsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$
end

3.2.1.3 Adam

The name *Adam* comes from *Adaptive Moment* and it can be described as the union of Momentum and RMS-prop, since is uses an exponentially weighted on a moving average momentum:

Algorithm 7: Adam Algorithm

Data: Learning rate ε ; Initial NN parameter θ
Data: Small constant ($\sim 10^{-8}$) δ ; decay rates ρ_1 and ρ_2
$\mathbf{r} = 0$
$\mathbf{s} = 0$
while *Stop criterion met* == *false* **do**
  Sample m-elements minibatch with inputs $\{\mathbf{x_1}, \mathbf{x_2}, , , , \mathbf{x_m}\}$ and targets $\{\hat{\mathbf{y_1}}, \hat{\mathbf{y_2}}, , , , \hat{\mathbf{y_m}}\}$
  $t \leftarrow t + 1$
  Gradient estimate: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L((\mathbf{x_i}; \theta)\hat{\mathbf{y}}_\mathbf{i})$
  Biased I ord moment: $\mathbf{s} \leftarrow \rho_2 \mathbf{s} + (1-\rho_1)\mathbf{g}$
  Biased II ord moment: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1-\rho_2)\mathbf{g} \odot \mathbf{g}$
  I ord bias correction: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\rho_1^t}$
  II ord bias correction: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-\rho_2^t}$
  Weights update:$\theta \leftarrow \theta - \varepsilon \frac{\hat{\mathbf{s}}}{\delta + \sqrt{\hat{\mathbf{r}}}} \odot \mathbf{g}$
end

3.3 Convolutional Neural Networks

Many grid-shaped inputs greatly benefit from the usage of *Convolutional Neural Networks* (CNN). As shown by groundbreaking results in image classification [25],

CNN outperform densely connected architecture whenever images are used as inputs. The reasons for this will be briefly explained is 3.3.2

3.3.1 Convolution operation

The convolution operation for a mono-dimensional, continuous function in time domain is defined as follows:

$$f * w(t) = \int_{-\infty}^{+\infty} x(a)w(t-a)da \tag{3.12}$$

Where f is the function the convolution is applied to and w is a function commonly referred to as *weighting function*. In this time domain example w must be 0 whenever $a < 0$ since it would take into account samplings made in the future but this only applies to this example and not to convolution in general. In the DL community f and w are often referred to as *input* and *kernel*, while the results is oftef referred to as *feature map*. In CNN convolution is applied to multi-dimensional discrete inputs (tensors) and it can be expressed as follows. In the case of 2D tensor input $\mathbf{I}$ and kernel $\mathbf{K}$:

$$\mathbf{I} * \mathbf{K}(i,j) = \sum_m \sum_n \mathbf{I}(m,n)\mathbf{K}(i-m, j-n) \tag{3.13}$$

The convolution operation is commutative, thus:

$$\mathbf{I} * \mathbf{K}(i,j) = \sum_m \sum_n \mathbf{I}(i-m, j-n)\mathbf{K}(m,n) \tag{3.14}$$

Cross-correlation

While the flipped indexes of kernel and filter guarantees the commutative property, that is usually not relevant in ML, and for sake of ease of implementation what is usually implemented is *cross-correlation*:

$$\mathbf{I} * \mathbf{K}(i,j) = \sum_m \sum_n \mathbf{I}(i+m, j+n)\mathbf{K}(m,n) \tag{3.15}$$

3.3.2 Benefits of using CNN

The reasons of CNN superior performance in image processing are briefly stated here:

Sparse connectivity

Since we want to detect small features (like edges) that occupy few pixels, the kernel is smaller than the input thus we store (and train) fewer parameter, and consequently perform fewer operations. Since each element of the input layer is connected to a few output layer element we call this *Sparse Connectivity*, as opposed to *dense connectivity*, where every element of the input layer is connected to every element of the output layer.

Parameter sharing

Parameter sharing using the same parameter than once. In a fully connected layer each parameter is used once when computing the output of layer. In CNN, each kernel element is used at every input position. This further reduces the memory footprint of the model to k parameters. Convolution is way more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency.

Equivariance

A function $f(x)$ is *equivariant* to a function g if $f(g(x)) = g(f(x))$. In the case of convolutional layers, be g any function that translates the input, then the convolution function is equivariant to g. When processing images, convolution creates a 2-D map of where certain features appear in the input. The object is moved in the input, its representation will move coherently in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations. CNN are not equivariant to other transformations, such as changes rescaling or rotation.

3.3.2.1 How CNN works

An intuitive explanation of the convolution operation is given here, using images as example. An RGB image such as the one in image 3.3 is *wxhx*3 tensor, where *w* and *h* are the the width and height respectively, while the filter is a *mxnx*3 tensor. The convolution can be seen as "passing" the filter $\mathbf{K}$ on the image $\mathbf{I}$. The first element is evaluated by overlapping the top left corners of he image and the kernel, operating a sum of the element by element product between the kernel and the portion of the image, as in 3.15 (in this case $i, j = 0, 0$). To evaluate the next element the kernel "slides" to the right by a number of pixels given by the *stride*. In a N-dimensional convolution, the stride is an N-dimensional vector whose element represent the entity of the sliding in each directions. 3.15 becomes:

$$\mathbf{I} * \mathbf{K}(i, j) = \sum_m \sum_n \mathbf{I}(is_1 + m, js_2 + n)\mathbf{K}(m, n) \tag{3.16}$$

With s_i i-th element of the stride vector. Each time the kernel slides a new element of the first row of the output is evaluated until the end of the input row is reached, then the kernel goes back to the left side of the image and down of s_2 pixels, and the second row of the output is evaluated just like the first one. This is repeated until the last row of the input is reached and we get the whole feature map.

- **Size**
  The the output of convolution operation is obviously smaller than the input, since the sliding stops m columns and n rows before the last ones. The size reduction can be avoided by using *zero-padding*, which means surrounding the input with a frame of zeros. This is particularly useful when we don't want to miss features on the border of the image. Obviously the size of the output is also reduced by the the stride. Given the 2-dimensional vectors $\mathbf{Is}, \mathbf{Ks}, \mathbf{S}$ and $\mathbf{P}$ (respectively Input-size, Kernel-size, Stride and Padding) the size of the t-th dimension is:

$$\left\lceil \frac{Is_t - Ks_t + 2P_t}{S_t} \right\rceil \tag{3.17}$$

- **Dimensions**
  In these example we always refer to a 2D-convolution, where the 2 dimensions are given by the 2 indexes in 3.15. The number of dimension of the convolution operation depends only on its indexes which, in the visual representation in figure 3.3, is the number of sliding directions of the filter. The dimensions of the input does not affect the dimensions of the convolution operation, but affect those of the kernel. A convolution with 5x5 kernel on a RGB image as in figure 3.3 would require $\mathbf{K}$ to be a 5x5x3 tensor, and the element-by-element product mentioned above would be between two 3-dimensional tensors. The red square in the input image in figure 3.3 is, in fact, a 3-dimensional matrix if we consider the RGB channels as its 3rd dimension. The same operation on a greyscale image would use a 5x5 tensor.
- **Depth**
  Each convolution operation between an input and a kernel generates a feature map, Usually each convolutional layer performs multiple convolutions on the input, so each layer has many kernels. All the kernels of a layer have the same size and stride, so that the dimensions of the feature maps are consistent, thus the output layer will be a 3-dimensional tensor whose first 2 dimensions are those of the feature maps, and the 3rd is the number of filters. The number of filters in a convolutional layer is equal to the 3rd dimension of the filter in the next layer.

Example

We apply two convolutional layers to the 260x125 RGB image in figure 3.3. Let the first layer be made of 32 8x8 filters with stride 4, and the second of 64 4x4 filters with stride 2.

Fig. 3.3: Visual representation of a convolution operation.

The first layer kernels are 8x8x3 tensors, thus the layer has $(8x8x3+1)x32$ weights (+1 because of the bias). The feature maps size are 63x30, so the 32 feature maps form a 63x30x32 tensor which is the input of the second convolution. The second layer kernels are 4x4x32 tensors, thus we have 32832 eights in this layer. The output of the second layer is a 30x13x64 tensor.

Chapter 4
Deep Reinforcement Learning Algorithms

As we mentioned in chapter 2, even though RL is extremely general, the vast majority of problems can not be brute-forced due to the dimensionality of action and spaces, that makes unfeasible to explore any state-action pair, the so-called "*Curse of Dimensionality*" [6]. For this reason Machine Learning techniques are typically used to find an optimal policy. When Deep Learning is used to solve a Reinforcement Learning control task we call it a Deep Reinforcement Learning Method.

In this chapter, after an overview on DRL methodologies classification, we present some of the most used DRL algorithms.

DRL formalization

The idea behind DRL is straightforward: we collect the state, action, reward tuples sampled by the agent and use this training set to train a NN whose objective is maximize the return; the updated policy is then used to draw new samples: if the return is above a certain threshold, the training is over, otherwise the new samples are used to train further.

This being said, while the procedure above might seem reasonable, it has a major flaw: all NN optimizer are based on gradient descent, thus we need to differentiate the objective function w.r.t NN gradients, but there is no direct correlation between the return and the policy. For this reason applying DL to RL is not straightforward. Different DRL algorithms overcome this issue is different ways, as will be shown later.

4.1 DRL algorithm classification

While only a few will be introduced in this work, there are several DRL algorithms, thus making necessary a classification.

Value Function Methods

To avoid the problem of optimizing a policy whose correlation with the objective function (the return G_t) is not explicit (more in 4.4), some methods completely avoid working directly on the policy, training an estimator for a VF instead. For example, having a finite number of actions and knowing $Q^{\pi}(s_t, a_t)$, the agent could pick the action that maximizes the expected return.

Policy Gradient Methods

These algorithms work directly on optimizing the policy by means of indirect correlations between the return and the policy. In this case the policy is a NN.

Actor-Critic Methods

Actor Critic methods are those DRL algorithms that make use of both NN policies and VF DL-based estimator.The VF estimator (the Critic) might be used, fir example, in the process of training the policy NN (the actor).

Off-Policy vs On-Policy

A method is called *on-policy* if the training set for the policy update is made of samples drawn with the policy obtained after the last update, while it is *off-policy* if the training set might also include samples taken with older policies. The advantage of off-policy is that samples can be stored and used later, making these methods less data-hungry.

4.2 DQN

DQN outline

As was mentioned in chapter 2, discrete action agents pick action from a finite set in any given state. For this reason, if we could evaluate the Q-function for all possible action, we could pick the action that maximize the expected return in any state, thus solving the task.

This was the idea behind the 2013 work by Mnih et al. [30], where a CNN is trained to estimate the Q-function and the agent uses the Q estimator to pick the action that maximizes it. Since a Deep Neural Network is used to estimate the Q-function, this method was called Deep Q Network (DQN). In the aforementioned application, the DQN solved a set of arcade games (from the Atari2600 console)

from raw pixel input (210x160). This method can be applied to any kind of MDP though, with the only limit of discrete action, since it would be unfeasible to evaluate the Q function for every action otherwise.

Fig. 4.1: Seaquest, one of the Atari2600 arcade video game solved by DQN.

4.2.1 DQN Implementation

Greedy Policy

To train the NN to estimate the Q-function of the state-action pairs, we should feed it enough state-action pairs, in other words, we should *explore*. According to the greedy policy, the agent undertakes that action that is expected to maximize the future reward, this would mean that the in a given state the agent will always take the same action preventing exploration. Of course, the Q estimator will change after after each update but this is not enough, especially considering the aforementioned bias in samples collection.

To ensure exploration DQN uses a ε-greedy policy, meaning that the agent takes a random action with probability ε, and the action that maximizes Q^{π} otherwise. The parameter ε represent the trade-off between exploration and exploitation; though being ~ 1 in the early stage of training, it decreases as the training goes on.

Replay Buffer

In principle, the most straightforward implementation of a Q-estimator based policy would sample a batch of transitions, use them once to update the NN and repeat the process until convergence is reached. However, this solution would present major flaws, that has been addressed using a different strategy, called *Replay Buffer*, consisting in storing transition in a large memory buffer, from which the training set is sampled at each NN update. This approach brings 3 advantages:

1. Data efficiency: each step of experience might be used several times.
2. Avoiding correlation: updating the policy from a set of consecutive timesteps, which are obviously correlated.
3. Braking bias loops: since current parameters determine the next train set, this kind of loops might get stuck in local minima or diverge badly.

Recalling the notion of Off-policy from 4.1, it can be noticed that DQN is an off-policy algorithm. In fact, the majority of steps sampled from the replay buffer was drawn with a policy older than the one that is about to be updated.

NN inputs and outputs

There are 2 possible ways of implementing a NN-based Q-function estimator:

1. A NN whose input is the state and the action, and whose output is the value of Q of the action fed as input.
2. A NN whose input is the state and whose output are the Q values of each possible action.

While the first one is closer to the Q-function formula, that takes state and action as inputs, the second one is more efficient, since it requires a single forward pass to find the action that maximize the expected return.

DQN algorithm

Here is reported the original algorithm by Mnih at al. [30].

Algorithm 8: DQN Algorithm [30]

Data: Learning rate ε ; Initial NN parameter θ
Data: Small constant ($\sim 10^{-7}$) δ
1 Initialize replay memory D to capacity N
2 Initialize action-value function Q with random weights
3 **for** *episode = 1, M* **do**
4 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\Phi_1 = \Phi(s_1)$
5 **for** *for t = 1, T* **do**
6 With probability ε select a random action a_t
7 Otherwise select $a_t = max_a Q^*(\Phi(s_t), a; \theta)$
8 Execute action a_t in emulator and observe reward r_t and image x_{t+1}
9 Set $s_{t+1} = s_t$, at, x_{t+1} and preprocess $\Phi_{t+1} = \Phi(s_{t+1})$
10 Store transition $(\Phi_t, a_t, r_t, \Phi_{t+1})$ in $\mathscr{D}$
11 Sample random minibatch of transitions $(\Phi_j, a_j, r_j, \Phi_{j+1})$ from $\mathscr{D}$
12 **if** Φ_{j+1} *is terminal* **then**
13 $y_j = r_j$
14 **else**
15 $y_j = r_j + \gamma max_{a'} Q(\Phi_{j+1}), a'; \theta)$
16 **end**
17 Perform a gradient descent step on $(y_j - Q(\Phi_j, a_j; \theta))^2$
18 **end**
19 **end**

Where

- Φ is the post-processed state. This is strictly related to the cited paper. Remember that $\Phi(x_t) = \Phi_t$
- $\mathscr{D}$ is the memory buffer. While not specified here, older samples are thrown off as new ones are memorized once the full capacity of the buffer is reached.

Note that at line 16 the Bellman Equation 2.8 applied to the State-Action Value function is used. The value of Q used to evaluate the estimator target y_i is computed by the estimator itself.

More recent implementation perform multiple gradient descent steps on larger batches; in other words, performing SGD on the Q estimator. To do this, they use a *target network* used to estimate y, in order to avoid having both the objective and the parameters varying during the NN update.

4.3 DDPG

DQN application is limited to discrete control problems for the very nature of DQN algorithm, since the output of the NN is the Q value associated with each possible action, and replicating this for a continuous control task is simply not possible.

To overcome this issue, the Deep Deterministic Policy Gradient (DDPG) [27] has been developed. DDPG is related to DQN since it relies on Q-function estimation, but in a different way.

Let us consider 2 distinct neural networks the first being μ, whose parameters are θ, and the second one being Q, whose parameters are ϕ. μ, called the actor, should output the optimal action given the state, and Q (the critic) the expected Q-value given the state and the action (note that this is an actor-critic algorithm). The Q estimator is different from the one seen in DQN (section 4.2): the action are in the input and the output is the expected Q for that particular action and state. This means that:

1. The Actor output can be fed to the Critic as part of the input (figure 4.2)
2. Is it possible to avaluate the gradient of Q w.r.t. the actions
3. Since the action are also the output of μ, is it possible to evaluate the gradient of Q w.r.t. θ

In other words, it is possible to optimize the Actor parameters to maximize the expected return.

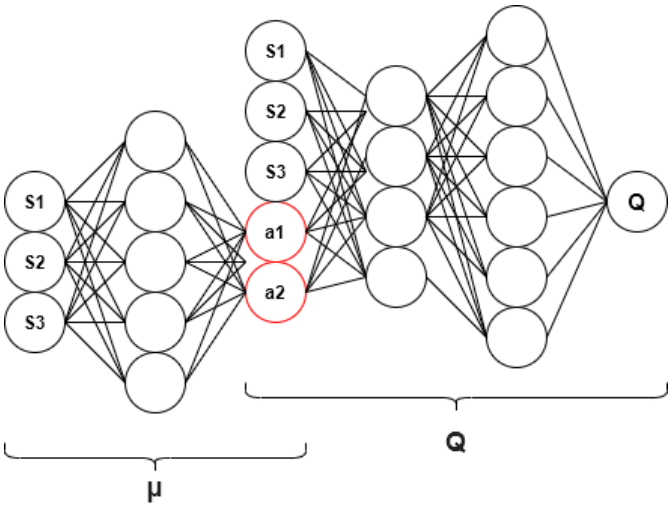


Fig. 4.2: Simple DNN Actor-Critic architecture.

4.3.1 DDPG algorithm

The algorithm pseudo-code from [27] looks as follows:

Algorithm 9: DDPG Algorithm

Data: Learning rate ε
Data: Random initialize Q and μ with parameter sets θ and ϕ
Data: Random initialize target NN Q' and μ' with parameter sets θ' and ϕ'
1 Initialize replay memory $\mathscr{R}$ to capacity N
2 **for** *episode = 1, M* **do**
3 observe initial state s_1 **for** *for t = 1, T* **do**
4 Select and Execut an action $a_t = \mu_\theta(s_t)$
5 Observe reward r_t and state s_{t+1}
6 Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathscr{R}$
7 Sample random minibatch of transitions (s_i, a_i, r_i, s_{i+1}) from $\mathscr{R}$
8 $y_i = r_i + \gamma Q'_{\phi'}(s_{i+1}, a'; \mu'_{\theta'}(s_{i+1}))$
9 Update critic: $L = \frac{1}{N}\sum_i (y_i - Q_\phi(s_i, a_i))^2$
10 Update actor: $\nabla_\theta J \approx \frac{1}{N}\sum_i \nabla_a Q_\phi(s,a)|_{s=s_i, a=a_i} \nabla_\theta \mu_\theta(s)|_{s_i}$
11 Update targets : $\phi' = \tau\phi + (1-\tau)\phi'$ and $\theta' = \tau\theta + (1-\tau)\theta'$
12 **end**
13 **end**

Where:

- There are 4 NNs: Actor, Critic, Actor Target and Critic Target.
- The target networks are needed to avoid dependency of the targets on the parameters.
- The (target) policy is used as (target) critic input to evaluate the objective at line 8.
- The expression at line 11 is similarly obtained, after differentiating the Q estimate w.r.t. θ: $\nabla_\theta J \approx \nabla_\theta \left[\frac{1}{N}\sum_i Q_\phi(s, \mu(s))|_{s=s_i, a=a_i}\right]$

4.4 Direct policy differentiation

In 4.1 we mentioned that the the correlation between the policy and the return is not explicit, otherwise the problem would be trivial, but this also mean that optimizing the policy such that the expected return increases is not straightforward. In DQN 4.2 the issue was completely avoided by estimating the Q value and picking the action that maximizes it, while in DDPG 4.3 we use chain differentiation on the expectation of Q to get the derivative w.r.t. policy parameters of the return. In neither case the policy is optimized directly because we don't know how to express the gradient of the objective w.r.t. the policy parameters. In order differentiate the objective J, being the sum of discounted reward, w.r.t policy parameters θ we need some additional consideration.

Recalling the definition of episode given in 2.2.2, we call *Trajectory* τ the set of state, action and reward tuples collected during the episode. Let us consider:

- π_θ, the policy described by the set of parameters θ. $\pi_\theta(a|s_t)$ is also the probability of taking the action a_t in s_t
- $p(s_{t+1}|s_t,a_t)$ is the transition probability. In a model-free algorithm we don't assume the knowledge of the transition probability.
- $p_\theta(s_1,a_1,s_2,a_2,...s_T,a_T)$, the trajectory probability, is the probability of a trajectory.

We write the trajectory probability in a more compact form

$$\pi_\theta(\tau) = p(s_1,a_1,s_2,a_2,...s_T,a_T) \tag{4.1}$$

In any given state s_t, the probability of taking the action a_t depends on the policy $\pi_\theta(a_t|s_t)$, while the probability of being in state s_{t+1} given the state and the action depends on the transition probability $p(s_{t+1}|s_t,a_t)$, so it is straightforward to write the trajectory probability as a chain of probability from initial state s_1 to terminal step T:

$$\pi_\theta(\tau) = p(s_1)\prod_{t=1}^{T} p(s_{t+1}|s_t,a_t)\pi_\theta(a|s_t) \tag{4.2}$$

The goal of direct policy differentiation is finding the set of parameters theta that maximize the objective:

$$\theta^* = argmax \quad \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_t r(s_t,a_t)\right] \tag{4.3}$$

The objective J of the optimization is

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_t r(s_t,a_t)\right] \approx \frac{1}{N}\sum_i\sum_t r(s_{i,t},ai_{,t}) \tag{4.4}$$

Where the objective is approximated by averaging over i trajectories. For brevity, we define:

$$r(\tau) = \sum_{t=1}^{T} r(s_t,a_t) \tag{4.5}$$

Recalling 4.2 we make the expectation explicit:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[r(\tau)] = \int \pi_\theta(\tau)r(\tau)d\tau \tag{4.6}$$

By differentiating 4.6 and remembering the derivative of the logarithm we obtain:

$$\nabla_\theta J(\theta) = \int \nabla_\theta\pi_\theta(\tau)r(\tau)d\tau = \int \pi_\theta(\tau)\nabla_\theta\log(\pi_\theta(\tau))r(\tau)d\tau \tag{4.7}$$

Then

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta\log(\pi_\theta(\tau))r(\tau)] \tag{4.8}$$

By applying 4.1 to 4.8 we obtain:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta\left(\log p(s_1) + \sum_{t=1}^{T}\log\pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t,a_t)\right) r(\tau)\right]$$

From 4.5 and considering that between the differentiated terms only the policy term derivative w.r.t. the parameters in non null:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\left(\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(a_t|s_t)\right)\left(\sum_{t=1}^{T} r(s_t,a_t)\right)\right] \tag{4.9}$$

The expected value is evaluated using Monte Carlo's method by averaging over N trajectories:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left[\left(\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t=1}^{T} r(s_{i,t},a_{i,t})\right)\right] \tag{4.10}$$

This allows to directly optimize the policy parameters just by sampling, without knowing the initial state probability, or the transition probability for both discrete and continuous control.

4.5 Simple PG Algorithm

From 4.10 we can obtain a simple RL algorithm:

Algorithm 10: Simple PG algorithm

Data: Learning rate α ; Initial NN parameter θ
Data: Number of sampling trajectories N
1 Initialize $\pi_\theta(a_t|s_t)$ with initial random θ
2 **for** *Policy update = 1, M* **do**
3 Sample $\tau_{i=1\ldots N}$ using policy $\pi_\theta(a_t|s_t)$
4 Estimate objective gradient
$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left[\left(\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t=1}^{T} r(s_{i,t},a_{i,t})\right)\right]$
5 Perform a gradient ascent step: $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
6 **end**

The sampling at line 3, also called *Policy Rollout* means letting the agent interact with the environment using the latest policy for multiple episodes while collecting all the s,a,r tuples in the process.The stopping criterion might be the the number of episodes or the accumulated number of steps. This process would be a double loop within the policy update loop.

An similar strategy would sample a single trajectory and use it to evaluate the gradient perform gradient ascent. In this case we would make more updates but the steps would be smaller, since the fewer the epochs the less accurate the MC estimate of $\nabla_\theta J$ is.

Pros of this method:

- Suitable for continuous control.
- Easy to understand and implement.
- Good behavior in in partially observable environments.
- Exploration ”for free”.

The exploration comes for free since the policy $\pi_\theta(a_t|s_t)$ is stochastic, thus exploring the state action space without the need of forcing exploration. Cons:

- On-Policy
- High Variance and noisy gradients.
- Slow convergence. Learning rate is hard to tune and larger batches are needed.

This algorithm is clearly on-policy: the latest policy is used for the rollout and *one* update, then the sampled are thrown away, since it is not possible to optimize the parameters from samples collected with a policy that used older parameters.

Also using a stochastic policy on complex and often stochastic environments increases the variance of the sum of reward. An higher variance in the sum of reward leads to a less accurate MC estimate of the objective. This uncertainty forces us to use larger training sets (more trajectories) and perform smaller updates (small α).

4.6 Generalized Advantage Function

The policy gradient as written in 4.8 is the most straightforward definition of PG, but there are several other possibilities, as shown in [35] . Given the general formulation:

$$g = \mathbb{E}\left[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{4.11}$$

Ψ_t can be:

- $\sum_{t=0}^{\infty} r_t$:reward of the trajectory
- $\sum_{t'=t}^{\infty} r_t$: reward following a_t or ”reward to go”
- $Q^\pi(s_t,a_t)$: state-action value function
- $A^\pi(a_t,s_t)$: advantage function

The latter formulas use the definitions:

$$V^\pi(s_t) := \mathbb{E}_{\substack{s_{t+1:\infty};\\ a_{t:\infty}}}\left[\sum_{l=0}^{\infty} r_{t+l}\right] \qquad Q^\pi(s_t,a_t) := \mathbb{E}_{\substack{s_{t+1:\infty};\\ a_{t+1:\infty}}}\left[\sum_{l=0}^{\infty} r_{t+l}\right]$$

$$A^\pi(s_t,a_t) := Q^\pi(s_t,a_t) - V^\pi(s_t)$$

Here we briefly recap what was mentioned in 2.2: the value function V^π measures how much a state is good (in terms of reward) following the policy π. The state-value function is slightly different, because evaluates the reward expected from taking the action a_t in state s_t, then following π. Looking at the definition of the advantage

function it appears that it is the expected difference between taking the action a_t and following the policy π in t. Previous quantities also have discounted versions, in which future reward are discounted by a γ factor.
Q^π, V^π and especially A^π quantities are not trivial to estimate but far more meaningful than the mere reward. In particular having a high advantage function value for a state-action tuple means that that action, taken in that states, performs better than the average (following the policy π). To estimate these quantities a function approximator, such as a NN) is needed. Reinforcement learning algorithm having two NN for Value function (critic) and policy (actor) are called actor-critic algorithm. It would be possible to directly fit a NN to approximate A^π, but since:

$$A^{\pi,\gamma} = E_{s^{t+1}}\left[r_t + \gamma V^{\pi,\gamma}(s_{t+1}) - V^{\pi,\gamma}(s_t)\right] = E_{s^{t+1}}\left[\delta_t^{V^{\pi,\gamma}}\right] \tag{4.12}$$

It is more convenient to approximate V and then calculate A, since V depends only on s_t, thus its variance is lower.
$\delta_t^{V^{\pi,\gamma}}$ is the temporal difference (TD) of the value function V following the policy π with future reward discount γ. The Generalized Advantage Function [35] evaluates the advantage function at each timestep using a sum of k terms

$$\hat{A}_t^{(k)} := \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = -V(s_t) + r_t + \gamma r_{t+1} + .. + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k}) \tag{4.13}$$

The generalized advantage estimator GAE(γ, λ) is defined as the λ exponentially-weighted average of these k-step estimators from:

$$\begin{aligned} \hat{A}_t^{GAE(\gamma,\lambda)} &:= \sum_{k=1}^{\infty} \lambda^{k-1} \hat{A}_t^{(k)} = (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + ...\right) \\ &= \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \end{aligned} \tag{4.14}$$

4.7 Proximal Policy Optimization

4.7.1 Off-policy learning and importance sampling

By default Policy Gradient is on-policy, which means that the policy does not change during sampling and policy parameters are updated only at the end of the batch of trajectories, as shown in Algorithm 10.

In principle, this could be avoided by using a technique called "Importance Sampling" (IS). IS means evaluating the expected value of a stochastic variable f(x) where x is distributed according to a distribution p using samples from another distribution q:

$$E_{x\sim p(x)}\left[f(x)\right] = E_{x\sim q(x)}\left[\frac{p(x)}{q(x)}(x)\right] \tag{4.15}$$

IS allows to calculate the gradient of the objective function with respect to the updated policy parameters θ' using samples taken with the old policy parameters θ. The objective function is:

$$J(\theta') = \sum_{t=1}^{T} E_{s_t\sim p_\theta(s_t)}\left[\frac{p_{\theta'}}{p_\theta} E_{a_t\sim\pi_\theta(a_t|s_t)}\left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} r(s_t,a_t)\right]\right]$$

The issue with this approach is in the $\frac{p_{\theta'}}{p_\theta}$ ratio, since it can neither be evaluated (the system dynamic is unknown in model-free methods) nor assumed to be ≈ 1.

4.7.2 Surrogate Objective Function

When using a symbolic differentiation tool to optimize the parameters of a NN we don't directly use the already-differentiated term of the loss. For this reason the loss implemented in DRL code (using the advantage function in 4.11) looks like:

$$J = \mathbb{E}\left[\log \pi_\theta(a_t|s_t) A^\pi(s_t,a_t)\right] \tag{4.16}$$

Similarly, starting from 4.11 we could also write, for chain rule:

$$\begin{aligned}\nabla_\theta J &= \mathbb{E}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) A^\pi(s_t,a_t)\right] \\ &= \mathbb{E}\left[\sum_{t=0}^{\infty} \frac{\nabla_\theta \pi_\theta(a_t|s_t)|_{\theta_{old}}}{\pi_{\theta_{old}}(a_t|s_t)} A^\pi(s_t,a_t)\right]\end{aligned} \tag{4.17}$$

Similarly for what stated for 4.16, it can be written that:

$$J = \mathbb{E}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A^\pi(s_t,a_t)\right] \tag{4.18}$$

The same result can be obtained by considering an Importance Sampling estimator of the advantage function ([24] [34]) in fact we can see the IS in 4.18. This will allow to use the same batch multiple times, event if after the first update, the policy will not be the same one used to collect the samples.

In addition, the ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is a measure of the divergence between the old and new policy: the more its norm is close to 1, the lesser the policy has changed.

Using an estimator for the Advantage Function, we obtain the so called *Surrogate Objective*:

$$\mathscr{L}_\pi(\pi) = E\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}(a_t|s_t)\right] \tag{4.19}$$

4.7.3 Kullback–Leibler divergence

The Kullback–Leibler divergence (KL-Divergence) [16] is a measure of how different two distributions are different. The KL Divergence of the distribution P with respect to the distribution Q is:

$$D_{KL}(P||Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} \tag{4.20}$$

Properties:

- $D_{KL}(P||Q) \geq 0$
- $D_{KL}(P||P) = 0$
- $D_{KL}(P||Q) \neq D_{KL}(Q||P)$

The KL Divergence of the new policy π' with respect to the older policy π is:

$$D_{KL}(\pi'||\pi)[s] = \sum_{a \in A} \pi'(a|s) \log \frac{\pi'(a|s)}{\pi(a|s)}$$

4.7.4 The Proximal Policy Optimization Algorithm

Proximal Policy Optimization [36] (PPO) algorithms enforce KL constraint in two ways:

1. KL Penalty [20] :
   Policy update solves unconstrained optimization problem. Penalty coefficient β_k changes between iterations to approximately enforce KL-divergence constraint.

$$\theta_{k+1} = \underset{\theta}{argmax} \left[\mathscr{L}_{\theta_k} - \beta_k D_{KL}(\theta||\theta_k) \right] \tag{4.21}$$

2. Clipped Objective [36] :
   Let the ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$. Then:

$$\mathscr{L}^{CLIP}_{\theta k}(\theta) = \underset{\tau \sim \pi_k}{E} \left[\sum_{t=0}^{\infty} \left[\min \left(r_t(\theta) \hat{A}^{\pi_k}_t, clip\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right) \right) \hat{A}^{\pi_k}_t \right] \right] \tag{4.22}$$

   Where $\varepsilon \approx 0.2$ Policy update is:

$$\theta_{k+1} = \underset{\theta}{argmax} \quad \mathscr{L}^{CLIP}_{\theta_k} \tag{4.23}$$

We inspect the meaning of the 2nd version of the objective, since it is used more often and its meaning is less evident.

First, consider that, being $\pi_\theta(a_t|s_t)$ a probability, its value is always in the 0,1 interval, thus $r_t \geq 0$. in addition, 4.22 is equivalent to:

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a),\ \ g(\varepsilon,A^{\pi_{\theta_k}}(s,a))\right) \tag{4.24}$$

where

$$g(\varepsilon,A) = \begin{cases} (1+\varepsilon)A\ A \geq 0 \\ (1-\varepsilon)A\ A < 0. \end{cases} \tag{4.25}$$

- **Positive Advantage**
  When the advantage is positive the objective becomes:

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)},(1+\varepsilon)\right)A^{\pi_{\theta_k}}(s,a). \tag{4.26}$$

  Because the advantage is positive, the objective will increase if the action becomes more likely (that is, if $\pi_\theta(a|s)$ increases). But the min in this term puts a limit upon the objective increment. For this reason the new policy does not benefit from going too far away from the old one.
- **Negative Advantage**
  When the advantage is positive the objective becomes:

$$L(s,a,\theta_k,\theta) = \max\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)},(1-\varepsilon)\right)A^{\pi_{\theta_k}}(s,a) \tag{4.27}$$

  This time, the objective will increase if the action becomes *less* likely. But the max in this term puts a limit to how much the objective can increase and again, the new policy does not benefit from going too far away from the old one.

  Pseudocode:

Algorithm 11: PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
$\theta_{old} = \theta_0$
for $iteration = 1, 2, ...$ **do**
  for $actor = 1, 2, ..., N$ **do**
    Run policy π_{old} in environment for T timesteps
    Compute rewards-to-go $\hat{R}_t = \sum_{i=t}^{\hat{T}} r_i \quad \hat{T} = min(T, t_{terminal})$
    Compute advantage estimates, $\hat{A}_t$ using 4.14.
  end for
  Optimize surrogate L w.r.t. θ, with K epochs and minibatch size $M \leq NT$ typically via stochastic gradient ascent with Adam.

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathscr{D}_k|T} \sum_{\tau \in \mathscr{D}_k} \sum_{t=0}^{T} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\varepsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathscr{D}_k|T} \sum_{\tau \in \mathscr{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

Is PPO on-policy?

It appears from algorithm 11 that since we perform multiple epochs on the policy, thus updating a policy which is different from the one used to sample from the environment, as it is also suggested by the IS estimator. This being said, PPO is still an on-policy method because the update (as a whole) is performed using data collected by rolling out the latest policy, and after the update the collected data are just deleted before the new rollout.

4.8 Imitation Learning

4.8.0.1 Motivation

DRL techniques allow, in principle, to learn to solve complex tasks from scratch, as shown in algorithms 8, 9, 10 and 11, where the training starts from randomly initialized NN. This approach might not always be preferable, since:

1. Expert trajectories, which are examples of the solved task, might be available, and include this knowledge would speed up the training.

2. Some tasks, especially the ones that require long term planning, are difficult to solve from scratch. Let us consider a robotic arm required to pick an object from a table and place it on a shelf: it would be extremely unlikely to randomly pick the right sequence of actions, and it is not trivial to define a continuous reward the can guide the agent in the right directions.

One of the possible ways of solving this is Imitation Learning (IL). IL uses samples from an expert to infer a policy. Typically, after the policy can imitate the expert, the agent is trained with RL to improve performance or, more often, generality.

4.8.0.2 Behavioral Cloning

The most straightforward example of of IL is *Behavioral Cloning* (BC). In BC we perform supervised learning on the policy π_θ, where the training set is given by the expert trajectories. We optimize the policy parameters θ such that given the same state, the probability to take the same action is maximum. To achieve so, the objective to minimize is the Mean Squared Error between the policy prescribed actions and the action taken by the expert.

Algorithm 12: Behavioral Cloning + RL

Data: Initial π parameter θ
Data: Training set of M timestep of expert trajectories $\hat{a}(s_t)$
for $epoch = 1, 2, ...$ **do**
  for $batch = 1, 2, ..., N$ **do**
    Sample S transition from the training set , $S = \frac{M}{N}$
    Fit the policy by regression on mean-squared error using a SGD method (such as Adams):

$$L = \frac{1}{S} \sum_{i=0}^{S} \left(\pi_\theta(s_i) - \hat{a}(s_i) \right)^2,$$

  end
end
Start RL training using θ to warm-start algorithm 11

Part II
Designing a framework for physics simulation in Machine Learning

Here we present some introductory concepts on constrained rigid body dynamics with contacts and inspect the synergies between physics simulations and deep learning and how we created a tool that can be used to train NN to solve robotic control tasks leveraging Multi Body Dynamics.

Chapter 5
The Project Chrono Multi-Physics Simulation Library

Here we give some introductory concepts on optimization-based numerical methods for the simulation of non-smooth rigid multibody dynamic. Smooth (a.k.a. penalty-based) methods allow compenetration between collision shapes, and the magnitude of the compenetration is proportional to the contact force reaction. In this approach the contacts are elastic forces between bodies and while this is conceptually straightforward, the CPU time is larger compared to non-smooth methods [29], since even smell compenetrations might lead to very large reaction forces, imposing very small timesteps. On the other hand, non-smooth methods offer better performance when simulating problems with contacts [45]. In non-smooth (NS) MBD we assume the contact shapes to be perfectly rigid and the contacts are considered unilateral constraints. This approach allows for larger timesteps at the price of a more complicated workflow that will be briefly introduced hereafter.

5.1 Mathematical Background

Here we mention some mathematical concepts necessary for the subsequent topic.

5.1.0.1 Measures

- A *measure* $\nu(\mathscr{E})$ is a function $\nu : \mathscr{E} \to \mathbb{R}$, where $\mathscr{E}$ is a set from a σ-algebra Σ of a measurable space (X, Σ), satisfying zero measure of empty subsets $\nu(\emptyset) = 0$ and countable additivity $\mu\left(\bigcup_{k=1}^{\infty} E_k\right) = \sum_{k=1}^{\infty} \mu(E_k)$.
- *Unsigned measures* $\nu : \mathscr{E} \to \mathbb{R}^+$.
- *Vector measures* $\nu : \mathscr{E} \to \mathbb{R}^n$.
- *Lebesgue measures* operate on $\mathscr{E} \subset \mathbb{R}^n$. For example the L. measure of intervals in $\mathbb{R}$ is the unsigned measure $\lambda_0([a,b]) = b - a$. Subsets of $\mathbb{R}^n$ might not be Lebesgue-measurable.

- *Borel measure* μ on a locally compact Hausdorff space $\mathscr{E}$, is any measure defined on the smallest σ-algebra containing the open sets of $\mathscr{E}$, which is the σ-algebra of the Borel sets.
- *Radon measures*:locally-finite Borel measures, s.t. $\forall$ point x of the measure space $\mathscr{E}$, $\exists$ an open neighbourhood $\setminus_x$ of x s.t. the measure of $\setminus_x$ is finite $\left|\mu(N_p)\right| < +\infty$. So $|\mu(\mathscr{C})| < +\infty \quad \forall \mathscr{C} \subset \mathscr{E}$.

5.1.0.2 Cones

- A set $\mathscr{K} \in \mathbb{R}^n$ is a n-dimensional *cone* if, $\forall\, \mathbf{x} \in \mathscr{K}$, $\beta\mathbf{x} \in \mathscr{K}\ \forall\, \beta \in \mathbb{R}^+$.

- Cones can be *convex*, *closed*, or *compact*, as happens for sets.
  If $\text{int}(\mathscr{K}) \neq \{\emptyset\} \rightarrow \mathscr{K}$ is *full*
  If $\mathscr{K}$ closed, convex and full $\rightarrow \mathscr{K}$ is *proper*
  $\mathscr{K} \cap -\mathscr{K} = \{\emptyset\} \rightarrow \mathscr{K}$ is *pointed* or *salient*

- *Second order cone* (Lorentz cone): a defined as

$$\mathscr{K} = \left\{(x_0, \mathbf{x}_1) \in \mathbb{R} \times \mathbb{R}^{p-1} : \|\mathbf{x}_1\|_2 \leq x_0\right\} = -\mathscr{K}^*. \tag{5.1}$$

  Lorentz cones are self-dual, self-scaled and symmetric.

- $\mathscr{K}^*$ is said to be the *dual cone* of $\mathscr{K}$ (a generic real vector space equipped with an inner product) if:

$$\mathscr{K}^* = \{\mathbf{y} \in \mathbb{R}^n : \langle \mathbf{y}, \mathbf{x} \rangle \geq 0 \quad \forall \mathbf{x} \in \mathscr{K}\}. \tag{5.2}$$

  $\mathscr{K}^*$ convexity is always verified with no further assumption on $\mathscr{K}$

- $\mathscr{K}^\circ$ is said to be the *polar cone*, opposite of the dual cone,if:

$$\mathscr{K}^\circ = \{\mathbf{y} \in \mathbb{R}^n : \langle \mathbf{y}, \mathbf{x} \rangle \leq 0 \quad \forall \mathbf{x} \in \mathscr{K}\} = -\mathscr{K}^*. \tag{5.3}$$

5.1.0.3 Variational Inequalities and CCP

5.1.1 Variational inequalities

In a DVI integrator, at least one VI has to be solved at each time step.

- A *Variational Inequality* VI: finding the solution $\mathbf{x} \in \mathscr{K}$ of:

$$\boxed{\mathbf{x} \in \mathscr{K} \quad : \quad \langle \mathbf{F}(\mathbf{x}), \mathbf{y} - \mathbf{x} \rangle \geq 0 \quad \forall \mathbf{y} \in \mathscr{K},} \tag{5.4}$$

Where $\mathscr{E}$ is a Banach space, $\mathscr{K}$ a closed and convex subset of $\mathscr{E}$ and $\mathbf{F}:\mathscr{K}\to\mathscr{E}$ a continuous functional ($\mathscr{E}$ being the dual space of E).
The solution of (5.4) is referred to as $\mathsf{SOL}(\mathscr{K},F)$.

- VI can equivalently be defined as:

$$\boxed{\mathbf{x}\in\mathscr{K} \quad : \quad -F(\mathbf{x})\in\mathscr{N}_{\mathscr{K}}(\mathbf{x})} \tag{5.5}$$

In certain cases of the solution of a VI can be proved to exist and be unique:

- $\mathbf{x}$ $\exists$ in 5.4 if $\mathscr{K}$ is *compact* (and convex).
- $\mathbf{x}$ $\exists$ in 5.4 if $F(\cdot)$ is *cohercive*, which means:

$$\frac{\langle F(\mathbf{x})-F(\mathbf{x}_0),\mathbf{x}-\mathbf{x}_0\rangle}{|\mathbf{x}-\mathbf{x}_0|}\to\infty \quad \text{as } |\mathbf{x}|\to\infty \tag{5.6}$$

- $\mathbf{x}$ $\exists!$ in 5.4 if $F(\cdot)$ is *monotone*:

$$\langle F(\mathbf{x})-F(\mathbf{x}_0),\mathbf{x}-\mathbf{x}_0\rangle>0 \quad \forall\mathbf{x},\mathbf{x}_0\in\mathscr{K} \tag{5.7}$$

Some optimization problems are sub-cases of VIs:

- *Nonlinear Complementarity Problem* (NCP): finding a $\mathbf{x}$ s.t.

$$\mathbf{F}(\mathbf{x})\geq\mathbf{0}, \quad \mathbf{x}\geq\mathbf{0}, \quad \langle\mathbf{F}(\mathbf{x}),\mathbf{x}\rangle=0, \tag{5.8}$$

Also, for for the sake of compactness:

$$\boxed{\mathbf{F}(\mathbf{x})\geq\mathbf{0} \quad \perp \quad \mathbf{x}\geq\mathbf{0},} \tag{5.9}$$

this is equivalent to a Variational Inequality where $\mathscr{K}=\mathbb{R}^n_+$:

$$\mathbf{x}\in\mathbb{R}^n_+ \quad : \quad \langle\mathbf{F}(\mathbf{x}),\mathbf{y}-\mathbf{x}\rangle\geq0 \quad \forall\mathbf{y}\in\mathbb{R}^n_+ \tag{5.10}$$

- *Linear Complementarity Problem* (LCP): finding $\mathbf{x}$ S.t.

$$\mathbf{A}\mathbf{x}-\mathbf{b}\geq\mathbf{0}, \quad \mathbf{x}\geq\mathbf{0}, \quad \langle\mathbf{A}\mathbf{x}-\mathbf{b},\mathbf{x}\rangle=0, \tag{5.11}$$

Also, for for the sake of compactness:

$$\boxed{\mathbf{A}\mathbf{x}-\mathbf{b}\geq\mathbf{0} \quad \perp \quad \mathbf{x}\geq\mathbf{0},} \tag{5.12}$$

this is equivalent to a VI where $\mathscr{K}=\mathbb{R}^n_+$ and with affine $\mathbf{F}$:

$$\mathbf{x}\in\mathbb{R}^n_+ \quad : \quad \langle\mathbf{A}\mathbf{x}-\mathbf{b},\mathbf{y}-\mathbf{x}\rangle\geq0 \quad \forall\mathbf{y}\in\mathbb{R}^n_+ \tag{5.13}$$

- *Cone Complementarity Problem* (CCP): finding $\mathbf{x}$ s.t.

$$\mathbf{A}\mathbf{x}-\mathbf{b}\in-\Upsilon^o,\quad \mathbf{x}\in\Upsilon,\quad \langle\mathbf{A}\mathbf{x}-\mathbf{b},\mathbf{x}\rangle=0, \tag{5.14}$$

Also, for for the sake of compactness:

$$\boxed{\mathbf{A}\mathbf{x}-\mathbf{b}\in-\Upsilon^o\quad\perp\quad\mathbf{x}\in\Upsilon,} \tag{5.15}$$

where Υ is a second-order Lorentz cone. The CCP is equivalent to a VI where $\mathscr{K}=\Upsilon$ and $\mathbf{F}$ is affine:

$$\mathbf{x}\in\Upsilon\quad:\quad\langle\mathbf{A}\mathbf{x}-\mathbf{b},\mathbf{y}-\mathbf{x}\rangle\geq 0\quad\forall\mathbf{y}\in\Upsilon \tag{5.16}$$

5.1.2 Differential problems

- An *Ordinary Differential Equation* (ODE) is a system

$$\frac{d\mathbf{x}}{dt}=\mathbf{f}(\mathbf{x},t) \tag{5.17}$$

  Where the initial value $\mathbf{x}(t_0)=\mathbf{x}_0$ is given. According to Cauchy-Lipschitz and Picard-Lindelhof theorems the solution $\mathbf{x}(0)$ $\exists!$ if $\mathbf{f}(\mathbf{x},t)$ uniformly Lipschitz continuous in $\mathbf{x}$ and continuous in t.
- *Differential Algebraic Equation* (DAE):

$$\mathbf{F}\left(\frac{d\mathbf{x}}{dt},\mathbf{x},t\right) \tag{5.18}$$

  In its *semi-implicit* form, that is as an ODE and algebraic constraints $\mathbf{g}$:

$$\frac{d\mathbf{x}}{dt}=\mathbf{f}(\mathbf{x},t) \tag{5.19a}$$

$$\mathbf{g}(\mathbf{x},t)=\mathbf{0} \tag{5.19b}$$

  Just like ODE, DAE need initial values $\mathbf{x}(t_0)=\mathbf{x}_0$.
- A *Differential Variational Inequality* (DVI) is:

$$\frac{d\mathbf{x}}{dt}=\mathbf{f}(\mathbf{x},\mathbf{u},t) \tag{5.20a}$$

$$\mathbf{u}\in\mathsf{SOL}(\mathbf{F},\mathscr{K}) \tag{5.20b}$$

  $\mathsf{SOL}(\mathbf{F},\mathscr{K})$ being is the solution to the VI$(\mathbf{F},\mathscr{K})$.
  DAE is a DVI sub-case: with n bilateral algebraic constraints one takes $\mathbf{F}$ as the vector of algebraic constraint residuals, and uses $\mathscr{K}=\mathbb{R}^n$ so that $\mathbf{F}=\mathbf{0}$ everywhere by definition VI.
- *Measure Differential Inclusion* (MDI)

$$\frac{d\mathbf{v}}{dt} \in \mathscr{K}(\mathbf{q},t) \tag{5.21}$$

$d\mathbf{v} = dt$ is a bounded variation function and $\mathscr{K}(\mathbf{q},t)$ is a set-valued function with closed graph and closed convex values. DVI can accommodate impulsive events.

5.2 System state

The system configuration at timestep t is given by m_q generalized coordinates $\mathbf{q}(t) \in \mathbb{R}^{m_q}$ while the velocity by the vector $\mathbf{v}(t) \in \mathbb{R}^{m_v}$.

It might happen, such as when we use quaternions for rotations, and angular velocities for velocities that $m_q \neq m_v$.

We express the i-th body configuration using the position $\mathbf{x}_i \in \mathbb{R}^3$ of its COG and its rotation matrix $\mathbf{A}_i \in \mathsf{SO3}$, both referred the absolute reference.

A matrix requires to store 3x3=9 scalars, so we parametrize rotations in $\mathsf{SO3}$ using S^3, the hypersphere of unit-length quaternions $\mathbb{H}_1$. The unit-quaternion representing the 3D rotation of the i-th body is $\rho_i \in \mathbb{H}_1$, a set of four scalars.

Rotation matrices and unit-length quaternions can be mutually converted.

5.2.1 Incremental update of state

Typically numerical methods to solve the Cauchy problems require to update the state as

$$\mathbf{q}^{(l+1)} = \mathbf{q}^{(l)} + \Delta t \mathbf{v}$$

with $h = \Delta t$ being the timestep. This is not straightforward when $m_q \neq m_v$.

Therefore, we need a way to get $\mathbf{q}^{(l+1)}$ from $\mathbf{q}^{(l)}$ and $h\mathbf{v}$. We leverage Lie algebras to accomplish this.

Considering the configuration $\mathbf{q}$ as an element of a Lie group G, then velocities $\mathbf{v}$ are the corresponding Lie algebra $\mathfrak{g} = T_I G$ (tangent at the smooth manifold G) and $\mathbf{v}(t) = \mathbf{p}^{-1}(t)\dot{\mathbf{p}}(t)$ for $\mathbf{v} \in \mathfrak{g}, \mathbf{q} \in G$.

The exponential map transforms Lie algebra elements into Lie group elements as $\exp : \mathfrak{g} \mapsto G$. Thus with local transformations in G it holds $\mathbf{p}(t) = \exp(t\mathbf{v})$.

To update the state we perform a *product*:

$$\mathbf{q}^{(l+1)} = \exp(h\mathbf{v})\mathbf{q}^{(l)}$$

The Lie algebra of $\mathbb{R}^n$ is $\mathbb{R}^n$ thus:

$$\mathbf{x}^{(l+1)} = \mathbf{x}^{(l)} + h\dot{\mathbf{x}} \tag{5.22}$$

The Lie algebra of unit quaternions is $\mathrm{Im}(\mathbb{H})$, while the exponential map is $\exp(\{0, \frac{1}{2}\omega h\})$. Considering that:

$$\exp(\rho) = \exp(\{a, \mathbf{b}\}) = e^a \left\{ \cos|\mathbf{b}|, \frac{\mathbf{b}}{|\mathbf{b}|} \sin|\mathbf{b}| \right\}$$

We have:

$$\begin{aligned} \exp\left(\left\{0, \frac{1}{2}\omega h\right\}\right) &= \exp\left(\left\{0, (\omega/|\omega|)\frac{1}{2}|\omega|h\right\}\right) \\ &= \left\{ \cos\frac{1}{2}|\omega|h, \frac{\omega}{|\omega|}\sin\frac{1}{2}|\omega|h \right\} \end{aligned} \tag{5.23}$$

So:

$$\rho^{(l+1)} = \exp\left(\left\{0, \frac{1}{2}\omega h\right\}\right)\rho^{(l)} \tag{5.24}$$

Therefore the configuration can be updated as follows:

$$\mathbf{q}^{(l+1)} = \exp(h\mathbf{v}^{(l)})\mathbf{q}^{(l)} \tag{5.25}$$

5.3 Constraints

Kinematic pairs (like revolute or prismatic joints) are referred to as bilateral constraints, and can be expressed using algebraic constraints.

We call the set $\mathscr{G}_{\mathscr{B}}$ of algebraic constraints:

$$C_i(\mathbf{q}, t) = 0 \quad \forall i \in \mathscr{G}_{\mathscr{B}} \tag{5.26}$$

And its jacobian: $\nabla_q C_i = [\partial C_i/\partial \mathbf{q}]^T$.

The time derivative of the constraint equations is evaluated as follows:

$$\frac{dC_i(\mathbf{q}, t)}{dt} = \frac{\partial C_i}{\partial \mathbf{q}}\dot{\mathbf{q}} + \frac{\partial C_i}{\partial t} \tag{5.27}$$

$$= \nabla_q C_i^T \dot{\mathbf{q}} + \frac{\partial C_i}{\partial t} \tag{5.28}$$

$$= \nabla_q C_i^T \Gamma(\mathbf{q})\mathbf{v} + \frac{\partial C_i}{\partial t} = 0 \tag{5.29}$$

$\nabla_q C_i^T \Gamma(\mathbf{q})$ will be abbreviated as ∇C_i^T. $\Gamma(\mathbf{q})$ is a linear map that is a identity except for the translational part and 4x3 blocks that transform angular velocities into quaternion derivatives : $\dot{\rho} = \frac{1}{2}\rho\{0, \omega^l\}$.

Each constraint has a lagrangian multiplier $\widehat{\gamma}_{\mathscr{B},i}$ s.t. the reaction force in absolute coordinates is $\widehat{\gamma}_{\mathscr{B},i}\nabla C_i^T$.

5.4 Contacts

Considering the bodies to be perfectly rigid (non-smooth contact), unilateral contacts lead to complementarity constraints.

We define the distance function per each contact pair as shown in Fig.5.1:

$$\Phi_i(\mathbf{q}) \geq 0 \tag{5.30}$$

and we assume it to be differentiable in $\mathbf{q}$.

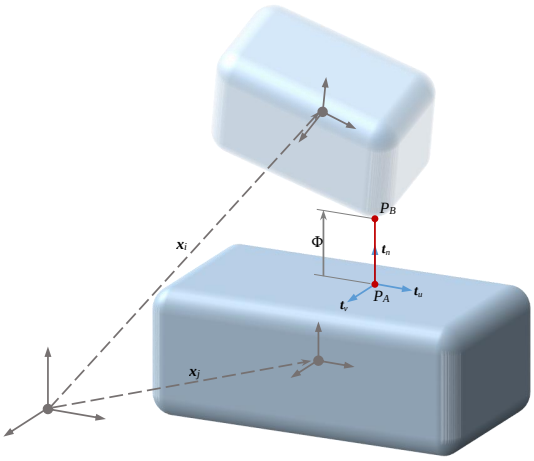


Fig. 5.1: The signed distance function for a couple of collision shapes.

The Signorini condition for a perfectly rigid (but frictionless) contact, leads to a complementarity constraint:

$$\Phi_i(\mathbf{q}) \geq 0 \perp \widehat{\gamma}_{n,i} \geq 0 \tag{5.31}$$

In other words, $\widehat{\gamma}_{n,i}$ is positive if distance is 0 (active contact), and viceversa distance is positive only if $\widehat{\gamma}_{n,i}$ is 0

We also consider a set of local coordinate systems centered in contact point with one normal $\mathbf{t}_{n,i} \in \mathbb{R}^3$ and two tangents $\mathbf{t}_{u,i}, \mathbf{t}_{v,i} \in \mathbb{R}^3$ axes, mutually orthogonal. The value of the normal force is expressed by a multiplier $\widehat{\gamma}_{n,i}$, while force multipliers $\widehat{\gamma}_{u,i}, \widehat{\gamma}_{v,i}$ express the tangential friction forces. We can write the contact force in 3D space as:

$$\begin{aligned}
\mathbf{F}_i &= \mathbf{F}_{n,i} + \mathbf{F}_{\parallel,i} && (5.32)\\
&= \mathbf{F}_{n,i} + \mathbf{F}_{u,i} + \mathbf{F}_{v,i} && (5.33)\\
&= \widehat{\gamma}_{n,i}\mathbf{t}_{n,i} + \widehat{\gamma}_{u,i}\mathbf{t}_{u,i} + \widehat{\gamma}_{v,i}\mathbf{t}_{v,i} && (5.34)
\end{aligned}$$

We also consider the velocities at the contact point, decomposed in normal $\mathbf{v}_{n,i}$ and tangential components $\mathbf{v}_{\|,i}$. They are related to absolute velocities $\mathbf{v} \in \mathbb{R}^{n_v}$ through the jacobians $\mathbf{D}_{n,i}, \mathbf{D}_{u,i}, \mathbf{D}_{v,i}$:

$$\begin{aligned}
\mathbf{v}_i &= \mathbf{v}_{n,i} + \mathbf{v}_{\|,i} && (5.35)\\
&= \mathbf{v}_{n,i} + \mathbf{v}_{u,i} + \mathbf{v}_{v,i} && (5.36)\\
&= u_{n,i}\mathbf{t}_{n,i} + u_{u,i}\mathbf{t}_{u,i} + u_{v,i}\mathbf{t}_{v,i} && (5.37)\\
&= (\mathbf{D}_{n,i}^T\mathbf{v})\mathbf{t}_{n,i} + (\mathbf{D}_{u,i}^T\mathbf{v})\mathbf{t}_{u,i} + (\mathbf{D}_{v,i}^T\mathbf{v})\mathbf{t}_{v,i} && (5.38)
\end{aligned}$$

The Coulomb-Amontons contact model μ_i and states that given the friction coefficient $\mu, \widehat{\gamma}_{n,i} \geq \sqrt{\widehat{\gamma}_{u,i}^2 + \widehat{\gamma}_{v,i}^2}$ for $\widehat{\gamma}_{n,i} \in \mathbb{R}^+$, and the tangential velocity at contact $\left\|\mathbf{v}_{\|}\right\|$ are in opposite direction, which means:

$$\left\langle \mathbf{F}_{\|}, \mathbf{v}_{\|} \right\rangle = -\left\|\mathbf{F}_{\|}\right\| \left\|\mathbf{v}_{\|}\right\|$$

Applying the Signorini condition 5.31, this frictional contact model becomes equivalent to an optimization constraint, expressed by the maximum dissipation principle [38, 39, 41]:

$$\begin{aligned}
\Phi_i(\mathbf{q}) \geq 0 \perp \widehat{\gamma}_{n,i} \geq 0 && (5.39)\\
(\widehat{\gamma}_u, \widehat{\gamma}_v) = \operatorname{argmin}_{\sqrt{\widehat{\gamma}_u^2 + \widehat{\gamma}_v^2} \leq \mu \widehat{\gamma}_n} \left(\widehat{\gamma}_u \mathbf{t}_1 + \widehat{\gamma}_v \mathbf{t}_2\right)^T \mathbf{v}_{\|}. && (5.40)
\end{aligned}$$

Please note that the contact model of Eq.5.40 depends only on a constant coefficient μ_i, therefore this formulation does do not make distinction between static and dynamic fiction as the original Coulomb-Amontons model does.

The Signorini condition can be expressed at the velocity level by recalling that $\dot{\Phi}_i(\mathbf{q}) = u_{n,i} = \mathbf{D}_{n,i}\mathbf{v}$:

$$\dot{\Phi}_i(\mathbf{q}) \geq 0 \perp \widehat{\gamma}_{n,i} \geq 0$$

The maximum dissipation principle of equation 5.40 can be developed for active contacts into a cone complementarity using the De Saxcé-Feng bipotential To this end one introduces second order Lorentz cones

$$\Upsilon_{\mathscr{A},i} = \left\{ \widehat{\gamma}_n, \widehat{\gamma}_u, \widehat{\gamma}_v \mid \mu \widehat{\gamma}_n \geq \sqrt{\widehat{\gamma}_u^2 + \widehat{\gamma}_v^2} \right\} \quad \subset \mathbb{R}^3$$

and their dual cones $\Upsilon_{\mathscr{A},i}^*$, so that equation5.40 can be written as a cone complementarity:

$$\widehat{\gamma}_i \in \Upsilon_{\mathscr{A},i} \perp \bar{\mathbf{u}}_i \in \Upsilon_{\mathscr{A},i}^*, \quad \forall i \in \{\mathscr{G}_{\mathscr{A}} | \Phi_i = 0\} \tag{5.41}$$

With:

$$\widehat{\gamma}_i = \left\{ \begin{array}{l} \widehat{\gamma}_{u,i} \\ \widehat{\gamma}_{v,i} \\ \widehat{\gamma}_{n,i} \end{array} \right\} \tag{5.42}$$

and

$$\bar{\mathbf{u}}_i = \left\{ \begin{array}{l} u_{n,i} + \mu_i \sqrt{u_{u,i}^2 + u_{v,i}^2} \\ u_{u,i} \\ u_{v,i} \end{array} \right\} \tag{5.43}$$

$$= \left\{ \begin{array}{l} u_{n,i} + \mu_i \left\| \mathbf{v}_{\parallel,i} \right\| \\ u_{u,i} \\ u_{v,i} \end{array} \right\} \tag{5.44}$$

$$= \mathbf{D}_i^T \mathbf{v} + \left\{ \begin{array}{l} \mu_i \left\| \mathbf{D}_{\parallel,i}^T \mathbf{v} \right\| \\ 0 \\ 0 \end{array} \right\} \tag{5.45}$$

$$= \mathbf{u}_i + \tilde{\mathbf{u}}_i \tag{5.46}$$

Where $\mathbf{D}_{\mathscr{A},i} \in \mathbb{R}^{m_v \times 3}$ and $\mathbf{D}_{\parallel,i} \in \mathbb{R}^{m_v \times 2}$, as:

$$\mathbf{D}_{\mathscr{A},i} = [\mathbf{D}_{n,i} | \mathbf{D}_{u,i} | \mathbf{D}_{v,i}] = \left[\mathbf{D}_{n,i} | \mathbf{D}_{\parallel,i} \right] \tag{5.47}$$

It is to be underlined that because of the $\tilde{\mathbf{u}}$ term, $\bar{\mathbf{u}}$ is a non-linear non-differentiable function of $\mathbf{v}$

$$\tilde{\mathbf{u}} = \left\{ \begin{array}{l} \mu_i \left\| \mathbf{v}_{\parallel,i} \right\| \\ 0 \\ 0 \end{array} \right\}$$

Note that the constraint of Eq.5.41 represents dissipative rule that is non-associated due to the the $\tilde{\mathbf{u}}$ term; without it would be associated [1].

Collisions introduce discontinuities, therefore to treat impulses we utilize vector signed Radon measures $d\gamma_i$ that, according to Lebesgue decomposition theorem can be decomposed as $d\gamma_i = \widehat{\gamma}_i(t)dt + \xi_i \, \widehat{\gamma}_i(t) \in L^1$: continuous forces over Lebesgue dt and impulses expressed by atomic measures ξ_i

[1] By constraining the velocity in the dual cone $\Upsilon^*_{\mathscr{A},i}$ we solve the *Associated Problem*. In principle, the velocity should be in the positive semi-space, but constraining it in the dual cone means that $\bar{\mathbf{u}}_i$ is in a cone that is narrower as the reaction cone is wider ($\Upsilon^*_{\mathscr{A},i}$ boundary is orthogonal to $\Upsilon_{\mathscr{A},i}$ boundary). We could either find $\tilde{\mathbf{u}}$ and correct the solution or neglect it and solve the associated problem, that will introduce a spurious overestimate of $u_{n,i}$

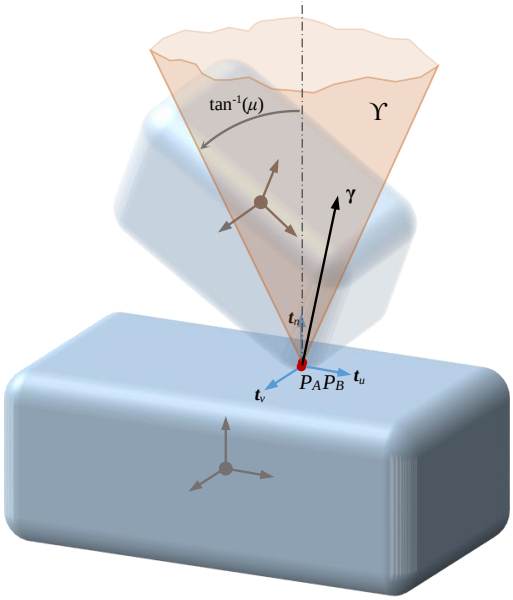


Fig. 5.2: The Coulomb friction cone for a single contact.

5.5 The dynamical model

The vector $\mathbf{f}(\mathbf{q},\mathbf{v},t) \in \mathbb{R}^{m_v}$ of generalized forces comprehends gravitational, external, applied gyroscopic forces, etc.

The mass matrix $\mathbf{M} \in \mathbb{R}^{m_q \times m_q}$ is block-diagonal and is composed of the masses and inertia tensors of the bodies.

If we neglected the velocity discontinuities, the MB model could be represented by this DVI:

$$\mathbf{M}\frac{d\mathbf{v}}{dt} = \mathbf{f}(\mathbf{q},\mathbf{v},t) + \sum_{i\in\mathscr{G}_\mathscr{A}} \mathbf{D}_{\mathscr{A},i}\widehat{\gamma}_{\mathscr{A},i}(t) + \sum_{i\in\mathscr{G}_\mathscr{B}} \nabla C_i \widehat{\gamma}_{\mathscr{B},i}(t) \tag{5.48a}$$

$$\widehat{\gamma}_{\mathscr{A},i} \in \Upsilon_{\mathscr{A},i} \perp \bar{\mathbf{u}}_i \in \Upsilon^*_{\mathscr{A},i} \quad \forall i \in \{\mathscr{G}_\mathscr{A} | \Phi_i = 0\} \tag{5.48b}$$

$$\widehat{\gamma}_{\mathscr{A},i} = \mathbf{0} \quad \forall i \in \{\mathscr{G}_\mathscr{A} | \Phi_i > 0\} \tag{5.48c}$$

$$C_i(\mathbf{q},t) = 0 \quad \forall i \in \mathscr{G}_\mathscr{B} \tag{5.48d}$$

$$\dot{\mathbf{q}} = \Gamma(\mathbf{q})\mathbf{v} \tag{5.48e}$$

Algebraic constraint are then differentiated to act at the velocity level [2], ths equation5.48d becomes:

$$\frac{dC_i(\mathbf{q},t)}{dt} = 0 \quad \forall i \in \mathscr{G}_{\mathscr{B}}.$$

Algebraic constraints can be reformulated to fit into the CCP:

$$dC_i(\mathbf{q},t)/dt \in \mathscr{Z}_{\mathscr{B},i} \tag{5.49}$$

where $\mathscr{Z}_{\mathscr{B},i} = \{0\}$ is a degenerate cone and $\mathscr{Z}^*_{\mathscr{B},i} = \mathscr{Z}^\circ_{\mathscr{B},i} = \mathbb{R}$, its dual. In fact, bilateral reactions $\widehat{\gamma}_{\mathscr{B},i} \in \mathbb{R}$.

The full set of bilateral constraints can be summarized as follows:

$$\bar{\mathbf{u}}_{\mathscr{B}} = [dC_1(\mathbf{q},t)/dt \mid ... \mid dC_{n_{\mathscr{B}}}(\mathbf{q},t)/dt] = \mathbf{0} \tag{5.50}$$

$$\widehat{\gamma}_{\mathscr{B}} = [\widehat{\gamma}_{\mathscr{B},1} \mid ... \mid \widehat{\gamma}_{\mathscr{B},n_{\mathscr{B}}}] \tag{5.51}$$

$$\mathbf{D}_{\mathscr{B}} = [\nabla \mathbf{C}_1(\mathbf{q},t) \mid ... \mid \nabla \mathbf{C}_{n_{\mathscr{B}}}(\mathbf{q},t)] \tag{5.52}$$

$$\Upsilon_{\mathscr{B}} = \bigtimes_{i \in \mathscr{G}_{\mathscr{B}}} \mathscr{Z}^*_{\mathscr{B},i} \tag{5.53}$$

$$\Upsilon^*_{\mathscr{B}} = \bigtimes_{i \in \mathscr{G}_{\mathscr{B}}} \mathscr{Z}_{\mathscr{B},i} \tag{5.54}$$

Therefore, the velocity level bilateral constraints can be expressed as a single CCP:

$$\widehat{\gamma}_{\mathscr{B}} \in \Upsilon_{\mathscr{B}} \perp \bar{\mathbf{u}}_{\mathscr{B}} \in \Upsilon^*_{\mathscr{B}}. \tag{5.55}$$

The same can be said for unilateral frictional contacts at Eq.5.48b-5.48c

We can only consider the active contacts and respective multipliers $\widehat{\gamma}_{\mathscr{A}*,i}$ introducing

$$\mathscr{G}_{\mathscr{A}*} = \{i \in \mathscr{G}_{\mathscr{A}*} | \Phi_i = 0\}$$

We can then adapt equation 5.48a to only consider active contacts $\sum_{i \in \mathscr{G}_{\mathscr{A}*}} \mathbf{D}_{\mathscr{A}*,i} \widehat{\gamma}_{\mathscr{A}*,i}(t)$ so 5.48b-5.48c can be rewritten as CCP:

$$\widehat{\gamma}_{\mathscr{A}*,i} \in \Upsilon_{\mathscr{A}*,i} \perp \bar{\mathbf{u}}_i \in \Upsilon^*_{\mathscr{A}*,i} \quad \forall i \in \mathscr{G}_{\mathscr{A}*} \tag{5.56}$$

From now on we will omit the $*$ asterisk for brevity, and we will use $\Upsilon_{\mathscr{A},i}$ for $\Upsilon_{\mathscr{A}*,i}$ and $\widehat{\gamma}_{\mathscr{A},i}$ for $\widehat{\gamma}_{\mathscr{A}*,i}$, always referring to active contacts only.

As we have done for bilateral constraint, we collect all frictional contacts:

[2] Solving the constraint at the position level introduces constraint drifting at the position level due to numerical errors. This calls for the introduction of stabilization terms

$$\bar{\mathbf{u}}_{\mathscr{A}} = [\bar{\mathbf{u}}_{\mathscr{A},1} \mid ... \mid \bar{\mathbf{u}}_{\mathscr{A},n_{\mathscr{A}}}] = \mathbf{0} \tag{5.57}$$

$$\widehat{\gamma}_{\mathscr{A}} = [\widehat{\gamma}_{\mathscr{A},1} \mid ... \mid \widehat{\gamma}_{\mathscr{A},n_{\mathscr{A}}}] \tag{5.58}$$

$$\mathbf{D}_{\mathscr{A}} = [\mathbf{D}_{\mathscr{A},1} \mid ... \mid \mathbf{D}_{\mathscr{A},n_{\mathscr{A}}}] \tag{5.59}$$

$$\Upsilon_{\mathscr{A}} = \bigtimes_{i \in \mathscr{G}_{\mathscr{A}}} \Upsilon_{\mathscr{A},i} \tag{5.60}$$

$$\Upsilon^*_{\mathscr{A}} = \bigtimes_{i \in \mathscr{G}_{\mathscr{A}}} \Upsilon^*_{\mathscr{A},i} \tag{5.61}$$

And rewrite them as a single CCP:

$$\widehat{\gamma}_{\mathscr{A}} \in \Upsilon_{\mathscr{A}} \perp \bar{\mathbf{u}}_{\mathscr{A}} \in \Upsilon^*_{\mathscr{A}}. \tag{5.62}$$

This allows to write all constraints (unilateral and bilateral) as a single system:

$$\mathbf{D}_{\mathscr{E}} = [\mathbf{D}_{\mathscr{A}} \mid \mathbf{D}_{\mathscr{B}}] \tag{5.63}$$

$$\widehat{\gamma}_{\mathscr{E}} = [\widehat{\gamma}_{\mathscr{A}} \mid \widehat{\gamma}_{\mathscr{B}}] \tag{5.64}$$

$$\bar{\mathbf{u}}_{\mathscr{E}} = [\bar{\mathbf{u}}_{\mathscr{A}} \mid \bar{\mathbf{u}}_{\mathscr{B}}] \tag{5.65}$$

$$\Upsilon_{\mathscr{E}} = \Upsilon_{\mathscr{A}} \times \Upsilon_{\mathscr{B}} \tag{5.66}$$

$$\Upsilon^*_{\mathscr{E}} = \Upsilon^*_{\mathscr{A}} \times \Upsilon^*_{\mathscr{B}} \tag{5.67}$$

The Multi Body model of equation 5.48 can be written as a compact DVI:

$$\boxed{\begin{aligned} \mathbf{M}\frac{d\mathbf{v}}{dt} &= \mathbf{f}(\mathbf{q},\mathbf{v},t) + \mathbf{D}_{\mathscr{E}}\widehat{\gamma}_{\mathscr{E}} && \text{(5.68a)} \\ \widehat{\gamma}_{\mathscr{E}} &\in \Upsilon_{\mathscr{E}} \perp \bar{\mathbf{u}}_{\mathscr{E}} \in \Upsilon^*_{\mathscr{E}} && \text{(5.68b)} \\ \dot{\mathbf{q}} &= \Gamma(\mathbf{q})\mathbf{v} && \text{(5.68c)} \end{aligned}}$$

The aquation 5.68b CCP is a VI($\mathbf{F}, \Upsilon_{\mathscr{E}}$) as in Eq.5.4 with $\mathbf{F} = \bar{\mathbf{u}}_{\mathscr{E}}(\gamma_{\mathscr{E}})$ nonlinear.

It is worth mentioning that removing the unilateral contacts a simpler special sub-case: semi-implicit DAE 5.19

$$\mathbf{M}\frac{d\mathbf{v}}{dt} = \mathbf{f}(\mathbf{q},\mathbf{v},t) + \mathbf{D}_{\mathscr{B}}\widehat{\gamma}_{\mathscr{B}}(t) \tag{5.69a}$$

$$\mathbf{C}(\mathbf{q},t) = \mathbf{0} \tag{5.69b}$$

In order to solve 5.48 or 5.68 at discrete timesteps we need a method that allows *discontinuities in velocities*. This is done by considering the problem as a Measure Differential Inclusion (MDI), making the assumption that velocities are just functions of bounded variations [42], and reaction forces can be discontinuous too.

5.6 Non-smooth dynamics

According to the Lebesgue decomposition theorem for each pair of signed measures ν and μ $\exists$ two signed measures s.t. $\nu = \nu_c + \nu_s$, where ν_c is absolutely continuous respect to μ, and ν_s,μ are singular. In this case μ is the time measure dt.

The Radon vector signed measures in this discussion are decomposed as follows

- the speed differential:

$$d\mathbf{v} = \mathbf{a}dt + \mathbf{j} \tag{5.70}$$

  We split the measure $d\mathbf{v} = \nu$ into an absolutely continuous part $\mathbf{a}dt$ w.r.t. to Lebesgue measure dt, and a pure point part $\mathbf{j}$ with null support responsible of velocity discontinuities. An impulse that causes a discontinuity at t_I will lead to $\mathbf{j} = (\mathbf{v}(t_I^+) - \mathbf{v}(t_I^-))$, where $\mathbf{v}(t_I^+)$ and $\mathbf{v}(t_I^-)$ are the left and right limits to t_I.
- the reaction impulse:

$$d\gamma = \widehat{\gamma}dt + \xi \tag{5.71}$$

  We split the measure $d\gamma$ into an absolutely continuous part $\widehat{\gamma}dt$ w.r.t. to Lebesgue measure dt and into a discrete part ξ (non-smooth contacts) whose dimension it that of a mechanical impulse.

  Reformulating equation5.68 DVI as a MDI leads to:

$$\boxed{\begin{aligned} \mathbf{M}d\mathbf{v} &= \mathbf{f}(\mathbf{q},\mathbf{v},t)dt + \mathbf{D}_{\mathscr{E}}d\gamma_{\mathscr{E}} && \text{(5.72a)}\\ d\gamma_{\mathscr{E}} &\in \Upsilon_{\mathscr{E}} \perp \bar{\mathbf{u}}_{\mathscr{E}} \in \Upsilon^*_{\mathscr{E}} && \text{(5.72b)}\\ \dot{\mathbf{q}} &= \Gamma(\mathbf{q})\mathbf{v} && \text{(5.72c)} \end{aligned}}$$

  See [40] for the proof of weak convergence of DI and the $h \downarrow 0$ convergence.

  In practice, solving the MDI leads to a time stepping scheme in which the unknowns are:

- velocity changes $(\mathbf{v}^{(t+h)} - \mathbf{v}^{(t)})$ in timestep h
- reaction impulses $\gamma = \int_{[t,t+h)} d\gamma$ in timestep h

5.7 The DVI time stepping method

Here is presented a method to perform the time integration no MDI-DVI equation5.72.

The timestepper introduced here is based on [41] work.

Equations 5.72 can be rewritten in discrete form in order to obtain a time-stepping problem from $t^{(l)}$ to $t^{(l+1)}$ [3]

[3] In smooth problems it boils down to $\gamma_{\mathscr{E}} = h\widehat{\gamma}_{\mathscr{E}}$

By recalling the impulses definition: $\gamma_{\mathscr{E}}$ such that $\gamma_{\mathscr{E}} = \int_{[t,t+h)} d\gamma$ and get the DVI/MDI time-stepping scheme:

$$\boxed{\begin{aligned} \gamma_{\mathscr{E}} \in \Upsilon_{\mathscr{E}} \perp \bar{\mathbf{u}}_{\mathscr{E}}^{(l+1)} \in \Upsilon_{\mathscr{E}}^* & \quad (5.73a) \\ \mathbf{M}^{(l)}(\mathbf{v}^{(l+1)} - \mathbf{v}^{(l)}) = \mathbf{f}(\mathbf{q}^{(l)}, \mathbf{v}^{(l)}, t^{(l)})h + \mathbf{D}_{\mathscr{E}}^{(l)} \gamma_{\mathscr{E}} & \quad (5.73b) \\ \mathbf{q}^{(l+1)} = \mathbf{q}^{(l)} + h\mathbf{v}^{(l+1)} & \quad (5.73c) \end{aligned}}$$

5.73 is composed of three main operations that are sequentially computed:

- The CCP of Eq.5.73a is solved for $\gamma_{\mathscr{E}}$ unknowns. Its solution is discussed below.
- 5.73b is a linear system of unknowns $\mathbf{v}^{(l+1)}$ that can be solved easily since $\mathbf{M}$ is block diagonal
- The configuration update $\mathbf{q}^{(l+1)}$ from equation 5.73c. The use of quaternions makes necessary the use of a the exponential map 5.25.

Solving the constraint at the velocity level means that inaccuracies might cause the constraints to drift and the contact shapes to penetrate, regardless of velocity constraint satisfaction.

To avoid this we introduce stabilization terms [3]:

$$\mathbf{b}_{\mathscr{A}} = \left[\frac{1}{h}\Phi_1 | 0 | 0 \mid ... \mid \frac{1}{h}\Phi_{n_{\mathscr{A}}} | 0 | 0 \right], \quad (5.74)$$

$$\mathbf{b}_{\mathscr{B}} = \left[\frac{1}{h}C_1 + \frac{\partial C_1}{\partial t} \mid ... \mid \frac{1}{h}C_{n_{\mathscr{A}}} + \frac{\partial C_{n_{\mathscr{A}}}}{\partial t} \right] \quad (5.75)$$

$$\mathbf{b}_{\mathscr{E}} = [\mathbf{b}_{\mathscr{A}} \mid \mathbf{b}_{\mathscr{B}}] \quad (5.76)$$

Therefore using a "corrected" version of the velocity $\underline{\bar{\mathbf{u}}}_{\mathscr{E}}^{(l+1)} = \bar{\mathbf{u}}_{\mathscr{E}}^{(l+1)} + \mathbf{b}_{\mathscr{E}}$, so equation 5.73a becomes

$$\gamma_{\mathscr{E}} \in \Upsilon_{\mathscr{E}} \perp \underline{\bar{\mathbf{u}}}_{\mathscr{E}}^{(l+1)} \in \Upsilon_{\mathscr{E}}^* \quad (5.77)$$

Or, equivalently:

$$\frac{1}{h}C_i + \nabla C_i^T \mathbf{v}^{(l+1)} + \frac{\partial C_i}{\partial t} = 0 \quad \forall i \in \mathscr{G}_{\mathscr{A}} \quad (5.78)$$

$$\frac{1}{h}\Phi_i + \nabla \Phi_i^T \mathbf{v}^{(l+1)} \geq 0 \quad \forall i \in \mathscr{G}_{\mathscr{B}} \quad (5.79)$$

for algebraic contacts and for the unilateral contacts respectively.[4].

[4] Actually, using directly $\frac{1}{h}\Phi_i$ can give problems when using very small time steps h, in fact if two bodies are inter-penetrating with $\Phi_i < 0$ because of inevitable integration errors, such stabilization term would cause an outbound separation speed that effectively will cancel the interpenetration gap at the next time step, but will also cause an unnatural 'popping' effect at the following time steps. To overcome this issue, we rather use the modified stabilization term:

We reorganize 5.73c such that it can be better processed. Local (contact and algebraic) constraint velocities can be expressed as function of generalized velocities (from eq. 5.43):$\underline{\bar{\mathbf{u}}}_{\mathscr{E}}^{(l+1)} = \underline{\bar{\mathbf{u}}}_{\mathscr{E}}(\mathbf{v}^{(l+1)})$, Moreover, from 5.73b we see that generalized velocities are function of reactions: $\mathbf{v}^{(l+1)} = \mathbf{v}(\gamma_{\mathscr{E}})$, so we want to get $\underline{\bar{\mathbf{u}}}_{\mathscr{E}}^{(l+1)} = \underline{\bar{\mathbf{u}}}_{\mathscr{E}}(\gamma_{\mathscr{E}})$.

Collecting:

$$\tilde{\mathbf{k}}^{(l)} = \mathbf{M}^{(\mathbf{l})}\mathbf{v}^{(l)} + h\mathbf{f}_t(\mathbf{q}^{(l)}, \mathbf{v}^{(l)}, t^{(l)}),$$

Left multiplying equation 5.73b by ${\mathbf{M}^{(\mathbf{l})}}^{-1}$:

$$\mathbf{v}^{(l+1)} = {\mathbf{M}^{(\mathbf{l})}}^{-1}\mathbf{D}_{\mathscr{E}}\gamma_{\mathscr{E}} + {\mathbf{M}^{(\mathbf{l})}}^{-1}\tilde{\mathbf{k}}. \tag{5.80}$$

Putting $\mathbf{v}^{(l+1)}$ as expressed in equation 5.80 in equation 5.43, we get:

$$\underline{\bar{\mathbf{u}}}_{\mathscr{E}}^{(l+1)} = \mathbf{D}_{\mathscr{E}}^T{\mathbf{M}^{(\mathbf{l})}}^{-1}\mathbf{D}_{\mathscr{E}}\gamma_{\mathscr{E}} + \mathbf{D}_{\mathscr{E}}^T{\mathbf{M}^{(\mathbf{l})}}^{-1}\tilde{\mathbf{k}} + \mathbf{b}_{\mathscr{E}} + \tilde{\mathbf{u}}_{\mathscr{E}}(\mathbf{v}^{(l+1)}) \tag{5.81}$$

The Delassus operator $\mathbf{N}$ and the vector $\mathbf{r}$ are introduced for compactness:

$$\mathbf{N} = \mathbf{D}_{\mathscr{E}}^T{\mathbf{M}^{(\mathbf{l})}}^{-1}\mathbf{D}_{\mathscr{E}} \tag{5.82}$$

$$\mathbf{r} = \mathbf{D}_{\mathscr{E}}^T{\mathbf{M}^{(\mathbf{l})}}^{-1}\tilde{\mathbf{k}} + \mathbf{b}_{\mathscr{E}} \tag{5.83}$$

Therefore, we have:

$$\underline{\bar{\mathbf{u}}}_{\mathscr{E}} = \mathbf{N}\gamma_{\mathscr{E}} + \mathbf{r} + \tilde{\mathbf{u}}_{\mathscr{E}}(\mathbf{v}^{(l+1)}) \tag{5.84}$$

Since $\tilde{\mathbf{u}}_{\mathscr{E}}(\mathbf{v}^{(l+1)})$ term is a non-linear function (of $\mathbf{v}^{(l+1)}$ (see 5.43), therefore also a nonlinear function of $\gamma_{\mathscr{E}}$, so equation5.77 becomes a NCP:

$$\gamma_{\mathscr{E}} \in \Upsilon_{\mathscr{E}} \perp \underline{\bar{\mathbf{u}}}(\gamma_{\mathscr{E}}) \in \Upsilon_{\mathscr{E}}^* \tag{5.85}$$

This implies two major downsides: it is difficult to both prove the existence of the solution and to numerically solve the NCP.

If $\tilde{\mathbf{u}}_{\mathscr{E}}$ is neglected the problem becomes convex [2], but this makes the contact model associated. This might build up a gap [4] proportional to $h\left\|\mathbf{v}_{i,\|}\right\|\mu_i$, but it does not grow further thanks to the stabilization term.

Neglecting the $\tilde{\mathbf{u}}$ term we therefore consider $\underline{\mathbf{u}}_{\mathscr{E}} = \mathbf{u}_{\mathscr{E}} + \mathbf{b}_{\mathscr{E}}$, a simplified version of $\underline{\bar{\mathbf{u}}}_{\mathscr{E}}$ term, that has the upside of being an affine function of $\gamma_{\mathscr{E}}$:

$$\underline{\mathbf{u}}_{\mathscr{E}} = \mathbf{N}\gamma_{\mathscr{E}} + \mathbf{r} \tag{5.86}$$

We clamp the $\frac{1}{h}\Phi_i$ to avoid unnatural "popping" when the body moves away from the interpenetration. The clamping threshold is known as *maximum speed of penetration recovery*

$$\max\{\frac{1}{h}\Phi_i, -\eta_s\}$$

This approach makes equation 5.77 solvable as a second-order convex CCP($\Upsilon_{\mathscr{E}}, \mathbf{N}, \mathbf{r}$):

$$\boxed{\gamma_{\mathscr{E}} \in \Upsilon_{\mathscr{E}} \perp \mathbf{N}\gamma_{\mathscr{E}} + \mathbf{r} \in \Upsilon_{\mathscr{E}}^*} \tag{5.87}$$

We recall (equation 5.16) that this CCP is equivalent to a VI with affine $\mathbf{F}$, thus (.5.5):

$$\gamma_{\mathscr{E}} \in \Upsilon_{\mathscr{E}} \quad : \quad \mathbf{N}\gamma_{\mathscr{E}} + \mathbf{r} \in -\mathscr{N}_{\Upsilon_{\mathscr{E}}}(\gamma_{\mathscr{E}})$$

This implies it is also equivalent to a convex program:

$$\boxed{\begin{aligned} \gamma_{\mathscr{E}} = &\text{argmin } \frac{1}{2}\gamma_{\mathscr{E}}^T \mathbf{N}\gamma_{\mathscr{E}} + \gamma_{\mathscr{E}}^T \mathbf{r} && \text{(5.88a)} \\ &\text{s.t.} \quad \gamma_{\mathscr{E}} \in \Upsilon_{\mathscr{E}} && \text{(5.88b)} \end{aligned}}$$

Project Chrono provides several numerical methods to solve the CCP in equation5.87 or 5.88.

- the fixed-point iteration (see [4]), being a variant of the Gauss-Seidell stationary iteration; this method is robust and is convenient to implement, but convergence might stall when dealing with chain of contacts(e.g. stacked bricks).
- To overcome these limitations addressed our research towards superior methods, such as P-SPG-FB, presented in [21], being a modification of the Spectral Projected Gradient method This method minimizes a function over separable convex constraints, exploiting the problem as in equation.5.88.
- A The Nesterov Accelerated Projected Gradient Descend (APGD)-based mehod, presented in [21] as well, performing closely to P-SPG-FB.

Finally, we rewrite 5.73b and 5.73a as a matrix expression that is not not used in the implementation but it is still useful to recapitulate:

$$\boxed{\begin{aligned} &\begin{bmatrix} \mathbf{M} & \mathbf{D}_{\mathscr{B}} & \mathbf{D}_{\mathscr{A}} \\ \mathbf{D}_{\mathscr{B}}^T & 0 & 0 \\ \mathbf{D}_{\mathscr{A}}^T & 0 & 0 \end{bmatrix} \begin{Bmatrix} \mathbf{v}^{(l+1)} \\ -\gamma_{\mathscr{B}} \\ -\gamma_{\mathscr{A}} \end{Bmatrix} - \begin{Bmatrix} \mathbf{M}\mathbf{v}^{(l)} + h\mathbf{f}^{(l)} \\ -\frac{1}{h}\mathbf{C}^{(l)} - \frac{\partial \mathbf{C}}{\partial t}^{(l)} \\ -\frac{1}{h}\boldsymbol{\phi}^{(l)} \end{Bmatrix} = \begin{Bmatrix} \mathbf{0} \\ \mathbf{0} \\ \underline{\mathbf{u}}_{\mathscr{A}} \end{Bmatrix} && \text{(5.89a)} \\ &\gamma_{\mathscr{A}} \in \Upsilon_{\mathscr{A}} \perp \underline{\mathbf{u}}_{\mathscr{A}} \in \Upsilon_{\mathscr{A}}^* \\ &\mathbf{q}^{(l+1)} = \mathbf{q}^{(l)} + h\mathbf{v}^{(l+1)} \end{aligned}}$$

5.8 Sensor Simulation

In simulation, we might ”cheat” and, being aware of the exact position, rotation, velocity and rotational velocity of any body and many other useful information, we can simple feed these to our control. This does not happen in reality. For a controller, the knowledge of the environment entirely depends on the typology and quality of its sensors.

There are 2 main reason for a dedicated sensor simulation model. Some sensors output information that the simulation can not provide. This is the case of camera sensors, to mention one. Other sensors, like GPS and IMU measure quantity that the simulation knows (being known as interoceptive), and their output is in fact based on simulation data. This being said, introducing a sensor model is necessary not just to simplify the code (all evaluations, such as position to GPS coordinates, happen under the hood) but to provide the level of noise and inaccuracy without which the information coming with the sensor output would be too accurate. Other sensors, such as robot encoder or vehicle speedometer, are accurate enough and physics-based, so there is no practical need to do anything different than simply taking the data from the simulation.

5.8.1 Sensor Simulated

- **IMU: Inertial Measurement Unit**
  Sensor based accelerometer, gyroscope and magnetometer used to report a body acceleration, orientation and angular rate.
- **GPS**
  GPS sensors are devices receiving a signal from the Global Positioning System satellites, reporting the coordinates of the sensor.
- **Camera**
  Camera sensors capture the electromagnetic radiation in the visible field in order to produce an image.
- **LIDAR: Laser Imaging Detection And Ranging**
  LIDAR sensors use laser distance measurement in order to get a point cloud that is the 3D representation of the targeted area.

5.8.2 Sensor Module Features

Chrono

Just like real sensors, sensors models can be linked to simulation bodies, such as a camera on the hood of the car.

Interoceptive sensors (GPS and IMU) make use of the physical quantities as previously mentioned.

GPU-Accelerated Ray Tracing

Exteroceptive sensing providing scene information about characteristics (camera and lidar). In Chrono::Sensor we rely on GPU-based ray tracing thanks to the OptiX library [32].

Camera sensor features lens models and post-processing noise augmentation while lidar, is based on [17].

update rate, time over which to collect data, and lag can be set for camera and LIDAR sensor as well.

Chapter 6
Simulating Multi Body dynamics in Python: The PyChrono project

Here we present the PyChrono project, a Python module providing the Python bindings for the C++ multi-physics library Project Chrono, featuring rigid and flexible Multi Body dynamics with contacts, real time 3D rendering, vehicle modelling tools and sensor simulation, and distributed as Conda package.

6.1 Motivation and context

6.1.1 The synergy between Simulation and ML

6.1.1.1 The need for data in ML-based control.

Machine Learning techniques (and DL in particular) learn from experience, thus requiring data to train. The size and quality of the dataset is crucial, and in every DL application, from medical images classification to language processing, a good dataset is a necessary condition to get good results, and robotics control makes no exception. The data collection in RL is built during the agent-environment interaction by memorizing the state-action-reward tuples. Training an agent in the real world from scratch is to some extent feasible, and it has been done successfully in the recent past [18] , while at the same time completely avoiding the issue of transferring in the real world the policy learned in simulation. This approach as severe drawback though:

1. **Safety:** except some cases, deploying a random agent on a real robot might be extremely dangerous for both the robot and whatever and whoever around it.
2. **Cost:** before starting any experiment it is needed to buy or build a real robot, with joints, actuators and sensor.
3. **Time:** Each robot can collect data for a single agent-environment interaction. This can be avoiding by deploying multiple robots, but this would further increase the costs.

For this reason, using simulation to collect data is extremely attractive, provided that:

1. Simulation should be as close as possible to reality, so that the policy learned in simulation can be transferred in the real world.
2. Simulations should also be as fast as possible, in order to decrease the time spent on collecting data.
3. The simulation can be spread across multiple workers to parallelize the samples collection process.

The first point is the most crucial in the sim-to-real policy transfer, since simulations cannot typically take into account neither the potentially unlimited variety of real world scenarios nor all the physics effects involved in a real system. These limits have been overcome *Domain Randomization*, which consist in randomly seep through various color and textures for the scenarios [47] or different values for physical parameters [31]. This being said, in many cases not considering physical effects might prevent from applying a policy learned in simulation to the real world (such as considering delay was found to be crucial in [44]. Another approach to ease the sim-to-real transition is to condensate pixel inputs in numerical states [11], but this is not always applicable, since it might not be possible to compress all the valuable information from perception in a vector. In general, simulations that are closer to the real world application, make the process of transferring the policy to the real world easier

Regarding simulation speed, modern Multi-Body simulation platform ([48] [15] [46]) are able to simulate constrained rigid body dynamics efficiently, considering also that simulation can almost always be parallelized. This being said, when multi-physics simulation in needed in order to take into account other effects, such as deformable bodies or fluid-solid interaction, computational cost increases to the point that it might not be feasible to collect samples through simulation.

Vehicle Dynamics Simulation

Simulating vehicle dynamics is conceptually identical to simulating constrained rigid body dynamics. This being said, modelling complex mechanical system such as cars is unpractical to do from scratch, like manually specifying mass properties, position, constraints etc of each component. For this reason, and considering that while being complex, wheeled vehicles share a common structure, platforms that are specialized in simulating vehicles provide vehicle templates to ease the modelling of the mechanical system. The level of detail of these templates varies from more simplified model [37] to higher level of details, allowing to simulate various types of tires, suspension, steering mechanisms etc.

Another valuable feature in Autonomous Driving-oriented simulation is providing driving scenarios, since training and testing ML agents in vast ans different scenarios is crucial given the nature of Autonomous Driving. Autonomous driving scenarios might include on-road or off-road scenarios, depending on the application.

6.1.1.2 Sensor Simulation

Robotic agents gather the information it needs through internal gauges (encoders, speedometers) and sensors used to interface with the surrounding. Often the low-dimensional Markovian state of the world is not available, thus we only have access to potentially high-dimensional on-board sensors, such as RGB cameras. While the kinematic state of the agents itself comes with the simulation itself, other sensors have to be simulated in order to replicate in simulation the information collection process of the real world deployment.

In the autonomy field there are various typologies of sensors involved, the most widespread (already introduced in 5.8) being:

- **IMU: Inertial Measurement Unit**
- **GPS**
- **Camera**
- **LIDAR: Laser Imaging Detection And Ranging**

The latter two pose peculiar challenges, since :

- The information they provide can not be directly simulated by a physics simulation library, thus needing an additional module or an interface with another tool.
- The memory footprint of the data produced by these sensor is greatly larger compared to other sensor. This has an effect both in the data flow during the simulation and on the size of the dataset.
- Camera sensors should also simulate real world undesired effects like lens distortion and flares.

Together with the data, sensor simulation should also be able to include update rate, lag and noise of the sensor implemented.

6.1.1.3 Advantages of a Python API

Interpreted Language

Python is an interpreted language, which means the code is executed line by line by an interpreter, without the need of creating an executable. This is especially welcome since it greatly eases and speeds up the modelling of mechanical systems, and modifications can be made without the need to compile a new executable. This also relieves the user from including headers and linking libraries to generate a makefile using a generator and running a compiler on the generated makefile, making the toolchain more streamlined.

Thanks to Python's package manager third-party packages are installed and can be imported in any script without any further action required. In addition, Python provides an interactive console to directly run commands and when debugging a

program is possible to use the console to interact with the variables stored by the program.

These features make Python comparatively easy to use and speed up the development of applications, and they are the reason why Python is so predominant in scientific and ML applications, but they come with a price: the execution speed of a Python scripts is vastly lower compered to the one of a binary file built from a compiled language such as C++ doing the same operations. While this might seem a major drawback that would prevent Python from being used in computationally intensive applications, this can be worked around, as will be shown in 6.2.

Bridging simulation and AI

As previously mentioned, the vast majority of ML framework, such as Tensorflow [1], PyTorch [33], Keras [14] and others provide Python API. In addition, Python is dominant in Data Science libraries as well, such as Pandas [50], and other third party packages offer support for scientific computing, linear algebra and plotting, respectively SciPy, NumPy [49] and MatPlotLib being the most used.

It goes without saying that implementing the functionalities provided by these libraries from scratch to use a different language is feasible only by a large team, and would require time to reach the level offered by the aforementioned packages, both in terms of completeness and in term of optimization and performance (all DL libraries make use of GPU parallelization).

The hegemony of Python in Machine Learning and Data Science makes providing a Python API a valuable feature for a simulation library. Let us consider the Reinforcement Learning process as represented in figure 2.2: at each step the agent has to send an action and receive a state and reward tuple from the environment (and a *done* boolean variable being true whenever the episode is over). This process, in terms of data flow, software architecture, code inspection and debugging, becomes way easier if everything happens within a single program, and this is only possible if both the ML libraries needed by the agent and the physics simulation and sensor simulation libraries needed by the virtual environment provide API for the same language. Otherwise there should be two different processes that exchange data, but this makes the task more challenging and prone to error for the developer, and introduce computational overhead due to inter-process communication. Being a sequential operation each process has to wait for the output of the other before proceeding, and exchanging large and complex data (such as tensor tuples) is both tedious and inefficient since the processes do not share memory.

6.2 The PyChrono project

For the reasons described in 6.1 a major effort has been put into a Python wrapper for Project Chrono that has the following requirements:

1. The Python bindings should include all classes and functions and not operate on an higher level, in order to provide a perfectly equivalent API.
2. It must expose the majority of Chrono modules so that users can use Multi Body Dynamics simulation, finite element analysis, vehicle dynamics, 3D rendering and more.
3. The overhead coming from the use of Python must be minimal
4. The data exchange must be efficient, especially when passing complex data coming from sensor simulation 5.8 .
5. Being a Python module, so that it can be imported and used by any script on a machine.
6. Being deployed distributed as a pre-compiled package to be installed by users without need for building pipeline (version control, generator, compiler).

The fulfillment of these requirements has brought to the creation of PyChrono, a Python module for physics and sensor simulation and 3D rendering whose implementations and features are discussed hereafter.

6.2.1 Python binding for a C++ library

6.2.1.1 Overcoming the computational efficiency issues of scripting languages

The numerical methods mentioned in 5 to simulate the behavior of Multi-Body constrained dynamics with contacts might be be entirely implemented in Python but, as aforementioned, Python is greatly slower compared to C++ and, due to the Global interpreter Lock (GIL), Python cannot leverage multithreading, since the interpreter is locked on one thread at a time. All this considered, it might seem illogical to use Python for scientific purposes in general, since they are typically computationally intensive.

The reason why Python is used with success in high performance computing applications is that python can be interfaced with compiled libraries through *Bindings*. Thus Python only acts as an high level interface to call the library function, that take care of the low level and computationally heavy part. For example, when simulating a mechanical system as in 5, the calculations needed to advance the physics happen inside the library, while the Python API allows to define the system more easily, calls the simulation stepping and, if the case, can directly pass data back and forth to the ML-based controller.

6.2.1.2 SWIG

To create a Python *Wrapper* for a C++ wrapper bindings for each class and function should be created. There are several software tools to automate this process, the most widespread being SWIG [5] (Simplified Wrapper and Interface Generator). SWIG provides tools to automatically generate the bindings between C/C++ code

and common scripting languages, such as Perl, C# and Python, without need to modify the underlying code.

SWIG uses a layered approach in which one layer is defined in C and other is defined in Python. The C layer contains low-level wrappers whereas Python code is used to define high-level features, since certain aspects of extension building are better accomplished in each language.

The *"Parser"* in the SWIG acronym means that it is capable of parsing C/C++ headers to automatically generate the double layer bindings. This being said, the generation process cannot be automated entirely, and some portion of the code require special attentions.

For this reason, SWIG wrapping process relies on interface files (.i) that specify both what to expose to Python, and how to manage what need to be handled specifically.

Interface files are scripts in which SWIG allows to both specify which classes and function should be exposed or to directly list the headers to be parsed to generate the bindings automatically.

In interface file we also take care of whatever need special attentions such as:

- Which classes should be treated as shared pointers. If a class is specified to be treated as a shared pointer all its inheritance tree must follow.
- How to instantiate templatic classes
- Defining *typemaps*: these SWIG directive change the input and the output of the C/C++ functions that have the same type and name of arguments as specified in the directive.
- Class extension: additional wrapper-defined methods to enhance the Python module, they can be defined either on the C++ or Python side.

Taking as example the following example of a SWIG interface file we can briefly explain how interface files work.

The following code itself does not define a Python module. The module script will include this file (%include ChMatrix.i) together with other containing the instructions to wrap other sections of the library. The following script wraps a class particularly complicated to expose to the Python API, and was chosen for this very reason, in order to show as much of the SWIG features as possible. The majority of classes do not require this amount of code (while often needing some particular attention), making the automated parsing of SWIG effective for wrapping large libraries as compared to tools that require the user to manually write the bindings for any class and function, such as PyBind.

In the interface file below we can notice:

- The target library headers are included to export the names of wrapped classes
- The typemap reported defines the maps for the GetMatrixData function (please note that the arguments types and names of the maps and the function are the same). This maps allows to change the behavior of the C++ function in order to return a list in which the content of a matrix is copied. This is a typical application of a SWIG typemap, because of the different ways to pass data in the two languages. In C++, as in GetMatrixData, to get an array of data it is allocated

outside the function and passed as an argument, while the ”Pythonic” way is to allocate inside the method and return it to the user. Let us see how the typemap works:

1. The function is called from Python using a single integer (the number of elements) as argument. In %typemap(in) the C++ function argument $1 and $2 are processed from the Python-side input $input. For example, whenever GetMatrixData is called for a 5x5 matrix, in the Python API the user will call GetMatrixData(25), and the %typemap(in) will allocate a 25 element double array p, an integer len and use them to call GetMatrixData(double* p, int len).
   Similarly, the %typemap(argout) takes care of the return of the Python function ($result). In this case, for example, the C++ function is void, while the Python function has to return a list populated with the matrix elements. To accomplish this, the %typemap(argout) instantiates a new Python List and populates it in a $len = \$2$ cycles for loop in which the i-th element of $p = \$1$ is assigned to the i-th element of the list.
   %typemap(freearg) frees the new memory allocated for $1 (while the list is passed to the user so the Python garbage collector will manage its memory).

- The class ChMatrixDynamic is defined in the interface code.
- Through %template macro we instantiate the templatic class ChMatrixDynamic. From the Python API users will use ChMatrixDynamicD class.
- The class extended by adding methods on the C++ side through the %extend SWIG macro. It is usually convenient to implement computationally intensive function on the C side.
- Through %pythoncode macro we add methods on the Python side. GetMatr just calls GetMatrixData after getting its dimension. The function is then added to the class using setattr. Using this double approach for the extension we can make the API more easy to use (the users does not have to specify the size of the matrix before getting its elements) while leveraging C++ to speed up the iteration (it wouldn't be possible to use the typemap mechanism without prior knowledge of the size of the array).
- %ignore Since ChMatrixDynamic is defined in ChMatrix.i, any other definitions will be ignored.
- %include This macro tells SWIG to parse the header and automatically create bindings for the classes and functions in it. SWIG will
  - Try to create bindings for everything in the header.
  - Create bindings for templatic classes for each template defined. If no template is specified, that class will be ignored.
  - Automatically manage inheritance. SWIG will ignore inheritance of parent classes that are not wrapped.
- The ”real” ChMatrix.i contains other classes and methods that haven't been reported for the sake of brevity.

```
%{
// File: ChMatrix.i
/* Includes the header in the wrapper code */
#include <math.h>
#include "chrono/core/ChMatrix.h"
#include <Eigen/Core>
#include <Eigen/Dense>
using namespace chrono;
%}
%typemap(in) (double* p, int len) %{
   if(!PyLong_Check($input))
      SWIG_exception(SWIG_TypeError, "expected integer");
   $2 = PyLong_AsUnsignedLong($input);
   $1 = new double[$2];
%}

%typemap(freearg) (double* p, int len) %{
   delete($1);
%}

%typemap(argout) (double* p, int len) {
   PyObject* list = PyList_New($2);
   int i;
   for(i = 0; i < $2; ++i)
      PyList_SET_ITEM(list, i, PyFloat_FromDouble($1[i]));
   $result = SWIG_Python_AppendOutput($result, list);
}

template <typename Real = double>
class ChMatrixDynamic : public Matrix<Real,Dynamic,
    Dynamic,RowMajor> {
  public:
    ChMatrixDynamic() : Matrix<Real, Dynamic, Dynamic,
        RowMajor>() {}
    ChMatrixDynamic(int r, int c) {
      Matrix<Real, Dynamic, Dynamic, RowMajor>();
      this->resize(r,c);
      }
    };

%template(ChMatrixDynamicD) :ChMatrixDynamic<double>;


%extend :ChMatrixDynamic<double>{
    public:

      double getitem(int i, int j) {
        return (*$self)(i, j);
        }

      void setitem(int i, int j, double v) {
        (*$self)(i, j) = v;
        }
```

```
        const int GetRows() {
          const int r = $self->rows();
          return r;
          }
        const int GetColumns() {
          const int c = $self->cols();
          return c;
          }

        void GetMatrixData(double* p, int len) {
          int r = $self->rows();
          int c = $self->cols();
          for (int i = 0; i < len; i++){
            int ri = floor (i/c);
            int ci = i - c*ri;
            p[i] = (double)(*$self)(ri, ci);

          }
        }

        void SetMatr(double *mat, int ros, int col) {
          ($self)->resize(ros, col);
          for (int i = 0; i < ros; i++){
            for (int j = 0; j < col; j++){
              (*$self)(i, j) = mat[i*col + j];
            }
          }

        }
      };

//
// ADD PYTHON CODE
//

%pythoncode %{


def GetMatr(self):
   cls = self.GetColumns()
   rws = self.GetRows()
   len = cls*rws
   lst = self.GetMatrixData(len)
   rs_list = reshape(lst, rws, cls)
   return rs_list

setattr(ChMatrixDynamicD, "GetMatr", GetMatr)

def __matr_setitem(self,index,vals):
   row = index[0];
   col = index[1];
   if row>=self.GetRows() or row <0:
```

```
        raise NameError('Bad row. Setting value at [{0},{1}] in a
            {2}x{3}
            matrix'.format(row,col,self.GetRows(),self.GetColumns()))
    if col>=self.GetColumns() or col <0:
        raise NameError('Bad column. Setting value at [{0},{1}] in
            a {2}x{3}
            matrix'.format(row,col,self.GetRows(),self.GetColumns()))
    self.setitem(index[0],index[1],vals)

setattr(ChMatrixDynamicD, "__setitem__", __matr_setitem)

%}
%ignore chrono::ChMatrixDynamic;
%include "../../chrono/core/ChMatrix.h"
```

Leveraging multiple CPU cores

Python Global Interpreter Lock constrains the Python interpreter to work on a single thread per process. Python does have a multithreading module, but it is still subject to the GIL, and its only purpose is to speed up I/O bound problems by leveraging concurrency. While it is still possible to use multiprocessing to leverage multiprocessing to spread the computational effort across multiple CPU cores, it is to consider that subprocesses don't share memory and communicate between them, or more often with the main process, by sending and receiving serialized objects. This makes multiprocessing attractive only when data are shared at lower frequencies, while in some applications, like linear algebra and parallelization of algorithm in general, the communication overhead introduced by multiprocessing makes more harm than good.

For this reason multithreading is the only viable options to leverage multiple CPU cores to speed up certain operations. While Python cannot *directly* use parallelization through multithreading, it can still benefit from it if it is leveraged by the libraries that are wrapped for Python. Linear Algebra Python packages such as NumPy can indeed use CPU multithreading (OpenMP) and ML frameworks such as PyTorch and Tensorflow, can leverage GPU parallel computing through CUDA.

Considering that Python is a high-level programming language whose philosophy is to rely on third-party binary libraries for computationally intensive tasks, the GIL limitations on multithreading is less important than it might seem.

In figure 6.1 we show how PyChrono can still use multiple CPU cores while being used from Python.

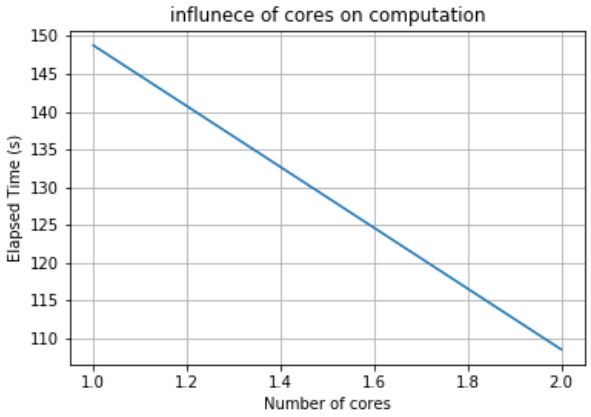


Fig. 6.1: Speedup in computation when parallelizing the task on 2 cores.

6.2.2 PyChrono Features

6.2.2.1 Physics Simulation Capabilities

The Chrono library is composed of several modules, and PyChrono replicate this modularity:

1. pychrono.core :
   The main module of PyChrono, providing the tools for Multi Body simulation.
   - Many-Body constrained dynamics with contacts (smooth and non-smooth)
   - Numerical tools: time integrators, solvers, functions
   - Actuators: linear and rotational; position, speed and torque/force controlled
   - Shafts: one-degree-of-freedom mechanical parts (gearboxes, planetary gears etc)
2. pychrono.fea
   Sub-module featuring deformable bodies simulation throug Finite Element Analysis. It provides:
   - 1D beams: Euler Beams, ANCF Beams, Isogeometric Cosserat Rod
   - 2D shells: Reissner shells, ANCF shells, Rotation-Free BST Shell Element
   - 3D elements: Tetrahedrons, Hexaedrons, Bricks
3. pychrono.irrlicht
   module for 3D rendering
   - Rigid and flexible bodies kinematics
   - Centers of Gravity, collision boundary boxes, collision shapes
   - FEA meshes

- Applied and contact forces
- Simulation data
- Human-in-the-loop simulation through GUI

4. pychrono.cascade
   Module for importing STEP files
5. pychrono.vehicle
   Module providing templates for modelling vehicles:

   - Tire Models: Pacejka Tire, Fiala Tire, FEA deformable tires, Lugre tires etc
   - Suspension Models: Double Wishbone, MacPherson, Multi Link amd more
   - Steering models: Pitman, Rack-pinion, Rotary arm
   - Powertrain models, Anti-roll Bar models, Driveline models
   - Terrain models: Flat/Hilly, Rigid/Deformable
   - Complete vehicle models: these are models already assembled that can be imported in the physical system

   In general the vehicle module offers an overall superior level of detail compared to the most widespread simulator for autonomous driving, still allowing various degree of model fidelity that trades off with simulation speed. Even the preexisting full vehicle models allow to tune the level of detail in both simulation and visualization.
6. pychrono.sensor
   Module providing sensor simulation, wrapping the functionalities described in 5.8:

   - Camera Sensor: RGB and Grayscale
   - Lidar sensor
   - GPS sensor
   - IMU sensor

7. pychrono.mkl
   Module that allows to use the Intel MKL Pardiso solver
8. pychrono.postprocessing
   Module that allows to postroprocess the simulation results, in particular to create POV-Ray videos.

6.2.2.2 Sensor Simulation in Python

In 5.8 we mentioned how Lidar and camers sensors are simulated on the GPU. The generated data pipeline needs special attentions since:

1. We want to avoid race conditions, such as the simulation trying to update the data while the user is accessing the same memory.
2. These data memory footprint is order of magnitude larger compared to state-based robotic observation (such as robotic arms state defined by 12 doubles).

While the second point importance is self-evident, the first one might be more obscure. The issue is that the sensor simulation process on the GPU runs independently from the simulation, thus their updates is not synchronized with the stepping of the environment. This might lead to race condition when the data fed to the control are accessed by the sensor simulation to be updated. For this very reason we developed this data pipeline:

1. To get data from a sensor we add a FilterAccess to it. We can add various types of filters to sensors, this in particular has a buffer member.
2. Each sensor stores data in its buffer member upon update. Buffers are array of structures, the type of structure varies according to the format of data being passed.
3. When GetBuffer() method is called the ownership of the buffer is passed (std::move) from the sensor to the FilterAccess buffer member.
4. We extended sensor with the HasData method to only get data when the buffer exists (in other words, if the latest buffer has already been moved, we don't want to move a null pointer to the filter's buffer)
5. Buffers are extended with a method that reinterpret arrays of structures as arrays of standard numeric types.
6. We use Numpy typemaps to convert raw arrays into numpy arrays.

It can be noticed how this strategy addresses the 2 concerns we mentioned. Having 2 different buffers prevents race conditions since the sensor new data will be written on the sensor buffer, while the user will work on the sensor buffer. Moving the buffer ownership is also efficient since data are only written once per sensor update, we only have to lock the buffer during the moving. Extending the Python API with the HadData method allows to check whether the sensor buffer is not a null pointer before moving it to the FilterAccess buffer. Finally, we have a safe access to the sensor data that can be casted to an array of a standard C++ numeric type, but it does not have a Python equivalent, thus we use typemaps similar to the ones seen in ChMatrix.i, but defined in a SWIG interface file called numpy.i distributed by NumPy developers. One of these typemaps, the Argout View Arrays, lets the C/C++ code provide the internal data and simply wraps the NumpPy classes around those raw arrays. While being the most critical typemap provided, since the memory of the array is still accessible by the C program, this is also the only one that allows to efficiently pass large data. Also, because of the double buffer strategy, the sensor will not write on the NumPy array memory. Please note that this approach only one, during the sensor update, while the ownership moving and the Argout View NumPy typemap do NOT instantiate new memory.

Below we show some of the interface code used to wrap the RGBA pixel buffer. The method getting the data is similar to the one used for the matrix, where the array is an argument in the C function while it is a return in the Python API.

The snippet below is just a part of the code needed to wrap a single sensor data buffer. Given the criticality of the data pipeline, the sensor module required a lot more hand-written interface code than average to wrap it.

```
// NumPy typemaps that convert a function taking pointers to an
    array an its dimensions as arguments into a Python function
    returning a 3 dimensional NumPy array.

%{
%apply ( uint8_t** ARGOUTVIEW_ARRAY3, int* DIM1, int* DIM2, int*
    DIM3) {(uint8_t** vec, int* h, int* w, int* c)};

////
////  PixelRGBA8 Extension
////
// Function returning False if the array of data (Buf of the
    BufferT class) is NULL

%extend
    chrono::sensor::SensorBufferT<std::shared_ptr<chrono::sensor::PixelRGBA8[]>>
    {
public:
bool HasData() {
   return !($self->Buffer==NULL);
}
};

%extend
    chrono::sensor::SensorBufferT<std::shared_ptr<chrono::sensor::PixelRGBA8[]>>
    {
public:
void GetRGBA8Data(uint8_t** vec, int* h, int* w, int* c) {
   *h = $self->Height;
   *w = $self->Width;
   *c = sizeof(PixelRGBA8)/sizeof(unsigned char);
   *vec = reinterpret_cast<uint8_t*>($self->Buffer.get());
}
};
```

6.2.3 Deployment of the Python package

Anaconda Distribution or Anaconda individual Edition (often simply Anaconda) is an open-source Python and R distribution, Anaconda Inc. also distributes two more complete (and not free) product edition: Teams and Enterprise edition. The purpose of Anaconda is to ease:

- The management of Python and Python packages versions, trough the creation of *virtual environments*. These are ”sealed” Python environment. A user can have multiple environments on the same machine.
- The packages installation and dependencies management through the Conda Package Manager. It installs Anaconda packages and their dependencies, avoiding package version conflicts in the meantime, something that Python package

manager (pip) can not do. This being said, pip packages can still be installed in Anaconda environments
- The creation of Anaconda Python packages, thanks to the *conda-build* tool.
- The distribution of Anaconda packages, through the Anaconda Cloud platform.

Anaconda also provides a GUI as alternative to the command line and a bootstrap version called Miniconda. In addition, Anaconda can be interfaced with PyCharm and Spyder, two of the most widespread Python IDE.

For these reasons we identified Anaconda packages as the right tool to deploy and distribute PyChrono.

6.2.3.1 Building the Conda Package

The building of a conda package required to call conda-build in a directory called *Recipe*, that contains:

- A metadata file meta.yaml where the information of the packages are written, such as package name, version, description, license, requirements for the builder and the user.
- Shell scripts (build.sh for Linux/MacOS and build.bat for Windows) that build and install the package
- An optional test file.

The Conda package has to be built for every Python version and Operative System, meaning to distribute the package for Python 3.6, 3.7 and 3.8 and for Windows Linux and MacOS the package has to be built 9 times. In addition, we wanted the package to be built on the newest code and distributed every time a new commit is pushed, or at least any time the Python API changes. For these reasons the only viable way to accomplish this is through a CI/CD (Continuous Integration Continuous Deployment) tool. CI/CD services allow to build, test and deploy the latest version of the code on several different OS on a single machine or on a web service running on a server.

At the beginning the CI of Project Chrono relied on Travis CI while the CD of Conda packages was on Appveyor CI. Later, due to the desire to have a leaner CI/CD pipeline, to the increasing importance of PyChrono in the Chrono Project and to the size of PyChrono, whose building process could not fit anymore in the time limit of Appveyor, the whole CI/CD has been moved to GitLab CI.

After the building and testing phase the Conda packages are built whenever a manual action is triggered (making this a Continuous Delivery rather than a continuous Deployment)

In order accomplish this the builder must have:

- Anaconda (Miniconda is sufficient) installed and activate for the shell in use.
- All the Conda packages needed by the building process (conda-build, SWIG, CMake, Jinja etc)
- Conda packages dependencies of the specific package

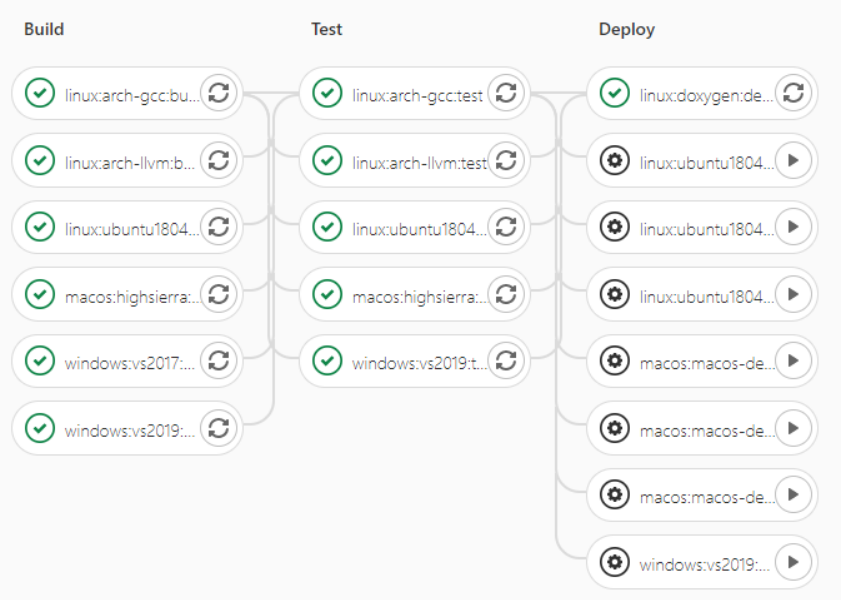


Fig. 6.2: The Chrono CI/CD pipeline. Pressing the Play button triggers the deployment of PyChrono.

- Not-Conda packages dependencies.

Some libraries needed by the builder can not be installed through Conda (such as Irrlicht in our case) but whenever it is possible to satisfy a dependency through Anaconda then using Conda is the best option since if specified in the metadata, conda packages needed by the users are installed automatically during the installation of the package.

Once the builder has everything it needs, we can launch the conda-build command on the recipe folder. Conda-build will use the source location from the metadata file, and then build according to the instructions in build.bat/.sh. We use CMake to generate, build and install the package. The installation is necessary for the *package relocation*, which is the ability of Conda to install the built package in any prefix (the prefix depends on Conda install location and the virtual environment name).

6.2.3.2 Distributing and Installing a Conda Package

Anaconda Cloud is a server that hosts Conda packages and from which users can download and install packages. Developers can create an account and upload their packages to their *Channel*. Channels upload is protected by login but it is possible to create a token (tokens are strings that can be used as an alternative to login) to upload conda packages from the command line. CI tools often allow to encrypt these

tokens. Using the token the Conda package can be uploaded to a specific Anaconda Cloud channel, projectchrono in this case, thus being available for the users. The users who have Anaconda installed can install PyChrono just by activating Conda and typing

```
conda install -c projectchrono pychrono
```

Obviously this CD tool transfer a lot of work from the users to the developers, since the whole pipeline has to be set up and after that it still require some maintenance, since new version of Python, of the building tools (such conda-build) or changes in the dependencies often lead to changes in the pre-build or build process.

This being said, distributing Conda packages makes the installation of PyChrono extremely easy and relieves the users of installing the whole versioning/generation/SWIG/building pipeline, since only Anaconda is needed. PyChrono users pool had great benefits from this, having reached more than 3000 downloads at the time of writing this document.

Part III
Applications

Side-by-side with the development of PyChrono, we used its simulation capabilities to create virtual environments of increasing simulation and control complexity. Hereafter we show the control taks that have been simulated using PyChrono and solved through DRL techniques.

Chapter 7
Implementation of tuple input capable a PPO Algorithm

In this chapter we will show and explain our custom implementation of the penalized version of the PPO algorithm. This implementation leverages PyTorch DL library and is capable of dealing with tuple observations.

7.1 Features

7.1.1 Tuple input definition and importance

DRL is known to work even with large and raw input tensors from its very beginning [30], and does not make any assumption on the type or architecture of the NN. This is especially welcome since the Markovian state of many problem requires to be expressed as a set of tensors of heterogeneous shape. A robotic manipulation tasks might require an RGB image (a 3D tesnor) but also the Propriocettivity (joint states and states derivatives, collected in a 1D tensor), a car to navigate needs images too, but also LIDAR point clouds (2D tensors) and data from speedometer, GPS, IMU, compass etc. that are collected in a single vector (1D tensor).

This being said, while being useful, tuple inputs are not supported by the most known DRL frameworks, whose algorithms can deal with single tensor inputs in the form of a 1D vector or a 3D RGB image.

For this reason we implemented our PPO algorithm, capable of dealing with tuple algorithms. It is to point out that while the algorithm itself does not make assumption on the shape of the tensors of the tuple, the NN does. For example, the NN suitable for tuple input in section 7.4 expects a tuple of 2 elements, whose first element is a RGB image and the second a vector. This being said, the modularity of the implementation allows to change only the NN to fit any new tensors shape without further modification to the rest of the code.

7.1.2 Leveraging Multiprocessing

The GIL prevents Python from distributing the computation over multiple cores, but we can still have multiple Python processes. This comes in handy during the Policy Rollout process, which is when we run the latest policy to collect new samples. To this extent, we rely on the SuProcVecEnv (Sub-Process Vectorized Environment) by OpenAI Baselines. This class contains a list to the environments running in different suprocesses, and takes care of the communication between the main process and the suprocesses where the simulations are running.

7.1.2.1 Saving and Resuming training

Since PPO is on-policy there is no replay buffer t save, and the PyTorch *Model State Dictionary* contains all the valuable information. The training can thus be saved and resumed easily with no loss.

7.2 Implementation

The implementation consists of 2 main files, ppo.py and Model.py, containing the PPO algorithm and various NN architecture respectively. The Python script we launch, train.py, is used to initialize the NN from Model.py and the environments from Gym (or one if its extension such as gym-chrono) and feed the to the algorithm together with the hyperparameters.

7.2.1 Hyperparameters

Hyperparameter tuning in DRL is crucial but also not generalizable, since different tasks might require different parameters. For this reason we set most of them arguments when launching the training. Here we pass general information, such as:

- Environment ID
- PyTorch model saving path

a NN architecture to use
e Number of parallel environments
u Terminal number of policy updates

But also the ADAM optimizer (algorithm 7) parameters:

l learning rate
s Timesteps between updates (together with parallel envs it defines batch size)
m Minibatch Size
p Epochs

7.2.2 Initialization

In the the train.py script we:

1. Initialize the Suprocess Environments
2. Initialize the PyTorch Model
3. Initialize the algorithm
4. Launch the training

7.2.2.1 Suprocess Environments Initialization

```
# Called in the main:
envs = SubprocVecEnv([make_env(env_name, i) for i in
    range(num_envs)])


def make_env(env_id, rank, seed=0):
   """
   Utility function for multiprocessed env.
   :param env_id: (str) the environment ID
   :param num_env: (int) the number of environments you wish to
       have in subprocesses
   :param seed: (int) the inital seed for RNG
   :param rank: (int) index of the subprocess
   """
   def _init():
      env = gym.make(env_id)
      env.seed(seed + rank)
      print(str(psutil.cpu_count(logical=False)))
      return env
   set_global_seeds(seed)
   return _init
```

The object passed to the algorithm as environment is a SubprocVecEnv from openAI Baselines. A list of openAI gym environments (gym-chrono in our case) is passed to it upon construction. This is done by make_env, that creates a gym environments each time is called in the list construction.

7.2.2.2 Model Initialization and resuming

```
model = ActorCritic([img_size[1], img_size[0]], sensor_size[0],
    num_outputs[0]).to(device)

   if os.path.isfile(modelpath):
      model.load_state_dict(torch.load(modelpath))
```

To be initialized the NN requires information about the shape of its input and the size of its output. If there's already a file in the path used to save the training, the algorithm loads the parameters to resume the training.

The training can now be launched:

```
ppo.ppo_train(num_steps, mini_batch_size, ppo_epochs,
              max_frames, max_pol_updates,save_interval,
                  increasing_length,
              test_interval, savepath=args.save_path)
```

7.2.3 Main Loop

This section of the code is the *ppo_train* function that is called from the main script. Until the maximum number of total timesteps (*frame_idx*) or updates (*max_pol_updates* is reached, this functions collects a dataset of samples and passes it to *ppo_update* to optimize the policy.

```
def ppo_train(self, num_steps, mini_batch_size, ppo_epochs = 8,
              max_frames = np.inf, max_pol_updates =
                  200,save_interval = 10, increasing_length = 0,
              test_interval = 40, savepath='./'):

   frame_idx = 0
   pol_updates = 0
   test_rewards = []
   state = self.envs.reset()
   mean_rewards = []

   while frame_idx < max_frames and pol_updates <
       max_pol_updates and not early_stop:
      log_probs = []
      values  = []
      states  = []
      actions = []
      rewards = []
      masks   = []
      entropy = 0

      for step_number in range(num_steps):
         if self.tuple_ob:
            arr_l = []
            for i in range(state.shape[1]):
               arr = np.stack( state[:,i][:] )
               arr_l.append(arr)

            state = [torch.FloatTensor(arr).to(self.device) for
                arr in arr_l]
         else:
```

```
            state = torch.FloatTensor(state).to(self.device)
        dist, value = self.model(state)

        action = dist.sample()
        next_state, reward, done, _ =
            self.envs.step(action.cpu().numpy())

        log_prob = dist.log_prob(action)
        entropy += dist.entropy().mean()

        log_probs.append(log_prob)
        values.append(value)
        rewards.append(torch.FloatTensor(reward).unsqueeze(1).to(self.device))
        masks.append(torch.FloatTensor(1 -
            done).unsqueeze(1).to(self.device))

        states.append(state)
        actions.append(action)

        state = next_state
        frame_idx += 1

    if frame_idx % test_interval == 0:
        mean_reward = Utils.CalcMeanRew(rewards, masks)
        mean_rewards.append(mean_reward)

    if self.tuple_ob:
        arr_l = []
        for i in range(next_state.shape[1]):
            arr = np.stack(next_state[:, i][:])
            arr_l.append(arr)

        next_state = [torch.FloatTensor(arr).to(self.device)
            for arr in arr_l]
    else:
        next_state =
            torch.FloatTensor(next_state).to(self.device)
    _, next_value = self.model(next_state)
    returns = self.compute_gae(next_value, rewards, masks,
        values)

    returns = torch.cat(returns).detach()
    log_probs = torch.cat(log_probs).detach()
    values  = torch.cat(values).detach()
    if self.tuple_ob:
        states  = Utils.cat_tuple_ob(states, state.shape[1])
    else:
        states  = torch.cat(states)
    actions = torch.cat(actions)
    advantage = returns - values

    self.ppo_update(ppo_epochs, mini_batch_size, states,
        actions, log_probs, returns, advantage)
    pol_updates += 1
```

```
    if (pol_updates)%save_interval == 0:
       torch.save(self.model.state_dict(), self.modelpath)
       print("Policy Saved after " + str(pol_updates) +
           "updates \n")
       print(datetime.datetime.now().time())
    np.save(savepath+'rew', np.asarray(mean_rewards))
```

In this section happens several things:

- All the counters are initialized and the environments are reset for the first time. The reset returns the initial states of the set of parallel environments that will be used afterwards.
- The policy update loop initialization, where all the dataset list are initialized
- The policy rollout. This for loop deserves a broader explanation below.
- In fixed intervals, we calculate the mean reward of the latest batch to monitor the training.
- We use the last states (next_state) to compute the last value functions through the model Critic.
- The PyTorch tensors are brought to the CPU. *Utils.cat_tuple_ob* is a method we implemented to manage the tuple observation that returns a list of list of tensors.
- The dataset is passed to the function performing the policy updates.
- The policy is saved every *pol_updates* updates and at the end of the training.

Tuple observation

In this section the tuple observation requires a special treatment 3 times (one of them under he hood, when detaching states). In the first 2 cases we are transforming a generic NumPy object whose shape is the number of parallel environment e and the tuple length l (so $[e,l]$). We reorganize it into a list of list of homogeneous arrays (*arr_l*), so that they can be converted into a list of tensors.

The policy rollout

The for loop in which the policy rollout happens is rather straightforward, given that the step function can be called over all the parallel environments simultaneously and advances all the environments. If one of this ends, it is reset (the end of the episode is still registered by the boolean variable done). All meaningful variables needed by the the training are are evaluated in the loop.

7.2.4 GAE

While the logarithmic probability and the entropy are evaluated using PyTorch distribution methods, estimating the Advantage Function through GAE 4.14 requires manual work. The evaluation is done backwards for efficiency reason. In fact, we can use the i-th GAE to evaluate the (i-1)-th GAE observing that:

$$\hat{A}_t^{GAE(\gamma,\lambda)} := \sum_{l=0}^{t_N} (\gamma\lambda)^l \delta_{t+l}^V = \delta_t^V + \gamma\lambda \hat{A}_{t+1}^{GAE(\gamma,\lambda)} \tag{7.1}$$

```
def compute_gae(self, next_value, rewards, masks, values,
    gamma=0.99, tau=0.95):
   values = values + [next_value]
   gae = 0
   returns = []
   for step in reversed(range(len(rewards))):
      # evaluate the n-ith Temporal Difference. The mask value
          is 0 whenever the step is terminal (done=True), so
          Vt+1 = 0
      delta = rewards[step] + gamma * values[step + 1] *
          masks[step] - values[step]
      gae = delta + gamma * tau * masks[step] * gae
      returns.insert(0, gae + values[step])
   return returns
```

7.2.4.1 Dataset Management

This function feeds a chunk of the dataset as large as the minibatch size each time is called. To do so, it creates a permutations of N integers to shuffle the data, splits them according to the minibatch size and yields the datasets data based on the set of indexes.

```
def ppo_iter(self, mini_batch_size, states, actions, log_probs,
    returns, advantage):
   batch_size = returns.size(0)
   ids = np.random.permutation(batch_size)
   ids = np.split(ids, batch_size // mini_batch_size)
   for i in range(len(ids)):
      if self.tuple_ob:
         yield [s[ids[i], :] for s in states], actions[ids[i],
             :], log_probs[ids[i], :], returns[ids[i], :],
             advantage[ids[i], :]
      else:
         yield states[ids[i], :], actions[ids[i], :],
             log_probs[ids[i], :], returns[ids[i], :],
             advantage[ids[i], :]
```

7.2.5 Update

Upon calling this method, per each epoch we iterate over the datset in minibatches thanks to *ppo_iter*. To get the objective function we evaluate the probability of the action in the newest policy to get the ration (r in 4.22). We use the exponential just because we have the logarithmic probabilities. Once this is done, we write the equation 4.22 to get the loss of the actor. In the equation it was an objective to maximize, this explains the minus before it.

The critic loss is simply the mean squared error between the expected VF (evaluated by the Critic) and the sampled VF.

An entropy term is added to stabilize.

Please note that the total number of optimizer steps per update is the number of minibatches times the number of epochs, and the number of minibatches is the number of steps per update times the number of parallel environment divided by the size of the minibatch.

The optimizer attribute of the ppo class is the PyTorch ADAM optimizer.

```
def ppo_update(self, ppo_epochs, mini_batch_size, states,
    actions, log_probs, returns, advantages, clip_param=0.2):
   for _ in range(ppo_epochs):
      for state, action, old_log_probs, return_, advantage in
          self.ppo_iter(mini_batch_size, states, actions,
          log_probs, returns, advantages):
         dist, value = self.model(state)
         entropy = dist.entropy().mean()
         new_log_probs = dist.log_prob(action)

         ratio = (new_log_probs - old_log_probs).exp()
         surr1 = ratio * advantage
         surr2 = torch.clamp(ratio, 1.0 - clip_param, 1.0 +
             clip_param) * advantage

         actor_loss = - torch.min(surr1, surr2).mean()
         critic_loss = (return_ - value).pow(2).mean()

         loss = 0.5 * critic_loss + actor_loss - 0.001 * entropy

         self.optimizer.zero_grad()
         loss.backward()
```

Please note that using as loss of the entire model the sum of the losses does not affect the optimization since the gradient of the critic loss w.r.t. the actor parameter is null and vice-versa.

7.3 Output and Exploration

PPO, like any Policy Gradient algorithm, makes use of a stochastic policy. This means that the policy, given the state, outputs a distribution of action probabilities. In the case of continuous actions we can use tanh output layer described in 3.1.1. The output in $(-1,1)$ can be scaled to the acceptable range of the physical problem and then the outputs are used as means of a multivariate gaussian distribution. The variances of the distributions are optimizable parameters the determine exploration vs exploitation: at the beginning of the training the variances are relatively large, as the training goes on the optimizer reduces their value, since taking actions closer to the means is on average more convenient as the policy performs better.

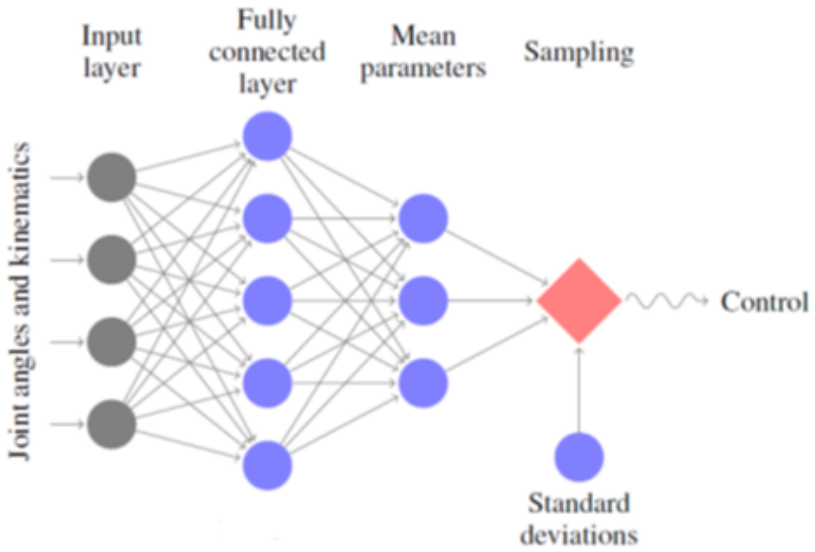


Fig. 7.1: The action is sampled from a Multivariate Gaussian whose means are the NN output

7.4 NN architectures

The algorithm explained in 7.2 imports a PyTorch Model about which very few assumptions are made:

- The output of the model must be in the form of a 2-elements tuple whose first element is a probability distribution and whose second element is a real number. Obviously, the actor-critic algorithm expects an actor-critic model.

- The model can process as input the state of the environment

To address the first requirement, all the models we provide contain 2 separate NN whose input is identical while the output of the actor has as many elements as the environment action and the critic has 1 output (the expected VF).

To match the input shape we implemented several NN architectures to accommodate vector inputs, RGB image inputs and tuple inputs.

To output a distribution, in the form of a multivariate Gaussian, each model also creates a 1D tensor of parameters whose dimension is the dimension of the output to be fed to the PyTorch Normal distribution constructor, in order to pass the distribution in the return. Since the variance is a Torch tensor it can be optimized to balance exploration and exploitation as mentioned in 7.3

Also parameters initialization done in *init_weights* is the same for every class in Model.py, although slightly different for Fully-Connected and Convolutional layers.

In all NN we use tanh activation for the last layer of each Actor to enforce the action in the $(-1,1)$ range, while all other activation use Rectified Linear Unit (ReLU).

```
def init_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.normal_(m.weight, mean=0., std=0.1)
        nn.init.constant_(m.bias, 0.1)
    if isinstance(m, nn.Conv2d):
        nn.init.xavier_normal_(m.weight)
        nn.init.constant_(m.bias, 0.0)
```

7.4.0.1 FF FCL Architecture

This is the simplest NN implemented. Please note:

- The size of inputs and outputs are not a constrain and are passed to the constructor.
- The Actor-Critic double NN, having the same input and the different output sizes.
- The forward method, implicitly called when calling the model, outputs a normal distribution and a number.

```
class ActorCritic(nn.Module):
    def __init__(self, num_inputs, num_outputs, hidden_size,
         std=0.0):
        super(ActorCritic, self).__init__()

        self.critic = nn.Sequential(
            nn.Linear(num_inputs, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, 1)
        )
```

```
        self.actor = nn.Sequential(
            nn.Linear(num_inputs, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, num_outputs),
            nn.Tanh()
        )
        self.log_std = nn.Parameter(torch.ones(1, num_outputs) *
            std)

        self.apply(init_weights)

    def forward(self, x):
        value = self.critic(x)
        mu = self.actor(x)
        std = self.log_std.exp().expand_as(mu)
        dist = Normal(mu, std)
        return dist, value
```

7.4.0.2 CNN Architecture

This architecture, taken from [30], is used to process RGB image observation, such as vision-only driving tasks. While preserving some common aspects with the previous NN, we can observe that:

- After 3 convolutional layers, the output of the last convolution is flattened and processed by a FCL into the final output. The flattening is a reshaping of the last convolutional feature done in *Flatten*.
- To flatten the last convolutional layers we must know its size. The size of the output feature of a convolution operation if evaluated according to equation 3.17. The *outputSize* function evaluates the size of the final output feature after an arbitrary number of convolution operations.
- The forward operations has to permute the input dimensions since RGB images come iwith the cannel (R, G and B) as last dimension, while PyTorch expects it to be the first.

```
def outputSize(in_size, kernel_size, stride, padding):
    conv_size = copy.deepcopy(in_size)
    for i in range(len(kernel_size)):
        conv_size[0] = int((conv_size[0] - kernel_size[i] +
            2*(padding[i])) / stride[i]) + 1
        conv_size[1] = int((conv_size[1] - kernel_size[i] +
            2*(padding[i])) / stride[i]) + 1

    return(conv_size)

class Flatten(torch.nn.Module):
    def forward(self, x):
        batch_size = x.shape[0]
```

```
        return x.view(batch_size, -1)

class ActorCritic_nature_cnn(ActorCritic):
    # CNN from Nature paper.
    def __init__(self, image_shape, num_outputs, std=-.5):
        super(ActorCritic, self).__init__()
        self.input_shape = image_shape
        fc_size = outputSize(image_shape, [8,4,3], [4,2,1],
            [0,0,0])
        self.actor = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU(),
            Flatten(),
            nn.Linear(fc_size[0] * fc_size[1] * 64, num_outputs),
            nn.Tanh()
        )

        self.critic = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU(),
            Flatten(),
            nn.Linear(fc_size[0] * fc_size[1] * 64, 1)
        )

        self.log_std = nn.Parameter(torch.ones(1, num_outputs) *
            std)

        self.apply(init_weights)


    def forward(self, x):
        # Input shape is [batch, Height, Width, RGB]
        # Torch wants [batch, RGB, Height, Width]. Must Permute
        x = ((x-127)/255).permute(0,3,1,2)
        value = self.critic(x)
        mu = self.actor(x)#.mul_(2)).add_(-1)
        std = self.log_std.exp().expand_as(mu)
        dist = Normal(mu, std)
        return dist, value
```

7.4.0.3 Tuple-Input architecture

In this NN the image is processed to a CNN as in the previous one, while the vector is passed to 1 hidden FC layer. Their output are oncatenated (*torch.cat*) and processed by 3 FC hidden layers.

The input is assumed to be a tuple of 2 elements, the first one being a 3D and the second one being a 1D tensor. There is no constrain on the size of the image or the vector, but different tuple would require a different NN architecture.

```
class MultiSensorEarlyFusion(nn.Module):
   def __init__(self, image_shape, sens2_shape, num_outputs,
       std=-0.5):
      super(MultiSensorEarlyFusion, self).__init__()
      self.input_shape = image_shape
      self.sens2_shape = sens2_shape
      self.num_outputs = num_outputs
      fc_size = outputSize(image_shape, [8, 4, 3], [4, 2, 1],
          [0, 0, 0])
      self.actor_cnn = nn.Sequential(
         nn.Conv2d(3, 32, kernel_size=8, stride=4),
         nn.ReLU(),
         nn.Conv2d(32, 64, kernel_size=4, stride=2),
         nn.ReLU(),
         nn.Conv2d(64, 64, kernel_size=3, stride=1),
         Flatten(),
         nn.ReLU(),
         nn.Linear(fc_size[0] * fc_size[1] * 64, num_outputs *
             5),

      )
      self.actor_fc0 = nn.Linear(sens2_shape, num_outputs * 5)
      self.actor_fc1 = nn.Linear(num_outputs * 10, num_outputs *
          20)
      self.actor_fc2 = nn.Linear(num_outputs * 20, num_outputs *
          10)
      self.actor_fc3 = nn.Linear(num_outputs * 10, num_outputs *
          5)
      self.actor_fc4 = nn.Linear(num_outputs * 5, num_outputs)

      self.critic_cnn = nn.Sequential(
         nn.Conv2d(3, 32, kernel_size=8, stride=4),
         nn.ReLU(),
         nn.Conv2d(32, 64, kernel_size=4, stride=2),
         nn.ReLU(),
         nn.Conv2d(64, 64, kernel_size=3, stride=1),
         Flatten(),
         nn.ReLU(),
         nn.Linear(fc_size[0] * fc_size[1] * 64, num_outputs *
             5),

      )
      self.critic_fc0 = nn.Linear(sens2_shape, num_outputs * 5)
```

```
        self.critic_fc1 = nn.Linear(num_outputs * 10, num_outputs
            * 20)
        self.critic_fc2 = nn.Linear(num_outputs * 20, num_outputs
            * 10)
        self.critic_fc3 = nn.Linear(num_outputs * 10, num_outputs
            * 5)
        self.critic_fc4 = nn.Linear(num_outputs * 5, 1)


        self.log_std = nn.Parameter(torch.ones(1, num_outputs) *
            std)
        self.apply(init_weights)


    def forward(self, data):
        x0 = ((data[0]-127)/255.).permute(0, 3, 1, 2)
        x1 = self.actor_cnn(x0)
        x2 = nn.functional.relu(self.actor_fc0(data[1]))
        x = torch.cat((x1, x2), dim=1)
        x = nn.functional.relu(self.actor_fc1(x))
        x = nn.functional.relu(self.actor_fc2(x))
        x = nn.functional.relu(self.actor_fc3(x))
        mu = torch.tanh(self.actor_fc4(x))
        std = self.log_std.exp().expand_as(mu)
        dist = Normal(mu, std)

        y1 = self.critic_cnn(x0)#.view(-1)
        y2 = nn.functional.relu(self.critic_fc0(data[1]))
        y = torch.cat((y1, y2), dim=1)
        y = nn.functional.relu(self.critic_fc1(y))
        y = nn.functional.relu(self.critic_fc2(y))
        y = nn.functional.relu(self.critic_fc3(y))
        value = self.critic_fc4(y)
        return dist, value
```

Chapter 8
Replicating and Solving Benchmark Environments

Here we show the environments that we provide as an open source alternative to benchmark continuous control algorithm and how we solved them, being the first application of PyChrono to DRL.

8.1 Introduction

The first step has been to replicate some of the OpenAI Gym MuJoCo environments. These environments are not meant to involve realistic robot model (as opposed to the Gym Robotics set of environments) but to provide benchmarks for continuous control algorithm that don't involve the processing of complex input types (such as raw camera pixels). The goal for this phase was to leverage the early stages of the PyChrono development demonstrating that it could be an open-source alternative for benchmarking DRL algorithms.

To this extent we replicated the InvertedPendulum and Ant environments of OpenAI gym using PyChrono to simulate physics and the algorithm described in 7 to solve them.

8.2 Environment, goal and Reward Shaping

8.2.1 Environments

8.2.1.1 chrono_pendulum-v0

Known as Inverted Pendulum, Reversed Pendulum, Cart-Pole or simply Pendulum. The environment consists of a sliding cart connected to a rod through an hinge. The

goal is to keep the pole upright and it is achieved by controlling the force applied to the cart.

It should be pointed out that this task does not require complex control techniques but it has been the first DRL application of PyChrono and it still is a useful tool to test the hyperparameters of an algorithm.

The Markovian state $s \in \mathbb{R}$ contains the position of the pole and the cart and the cart-pole angle and their rate of change (translational speed of the cart and relative rotational speed of the pole)

The actions vector is mono-dimensional, being the the horizontal force applied to the slider. It goes without saying that all observations and actions are continuous for this environment.

Every time the pole angle or the cart displacement reach a given limit value the environment is reset, meaning that the episode is aborted and the systems is brought back to the initial condition.

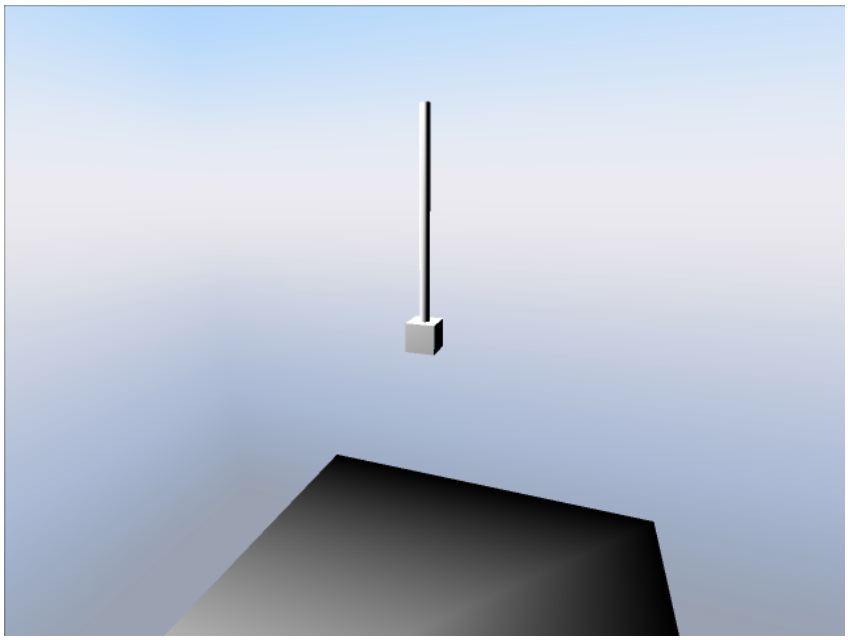


Fig. 8.1: The chrono pendulum-v0

Reward

Each timestep the agents gets a 1.0 reward, so the longer the episode lasts, the higher the sum of rewards will be. The reset conditions lead the control to keep the pole

almost vertical and the cart close to the initial position, otherwise the environment will reset and the sum of rewards will be low.

In other words we want the agent to learn to balance the pole just by rewarding the survival of the agent.

8.2.1.2 chrono_ant-v0

The ant is a physical model of a simplified 4-legged walker, even though it could be argued that ants, being insects, have 6 legs, but the name was given by the OpenAI gym environment we replicated.

Each leg is composed of a spherical central body linked to 4 legs by means of Revolute joints (1 DOF). Legs are slender cylinders and each one is linked to an ankle through another Revolute joint. Each ankle is the union of a slender cylinder and a semi-sphere (the ”foot”). The number of DOF relative to the central body is 8, while the total number of DOF is 14 because we use Euler Angles to describe the orientation of the sphere, since singularity conditions are avoided by the modest entity of the central body rotations.

The state s is a vector of 30 elements:

- the DOF and their derivatives (positions and velocities) except the planar position of the central body: $14 * 2 - 2 = 26$
- The feet-ground contact force: 4 elements. The forces are clipped to cut unnecessary spikes and since the agent only needs to know if the foot is touching the ground.

The action, on the other hand, is an 8-element vector, being the torques applied to the 8 leg joints.

The body capable of contacts are the ground (a big box), the central body (sphere) and the feet (4 spheres).

Reward

We want the walker to:

- **Advance as fast as possible**
  The walker is obviously supposed to walk. It should walk straight and as fast as possible.
- **Do not fall**
  The robot is considered fallen when the central body hits the ground. We obviously don’t want the walker to drag itself on the ground.
- **Avoid self-collision**
  Self-collisions is are collision between parts of the robot, such as feet colliding with the central body.

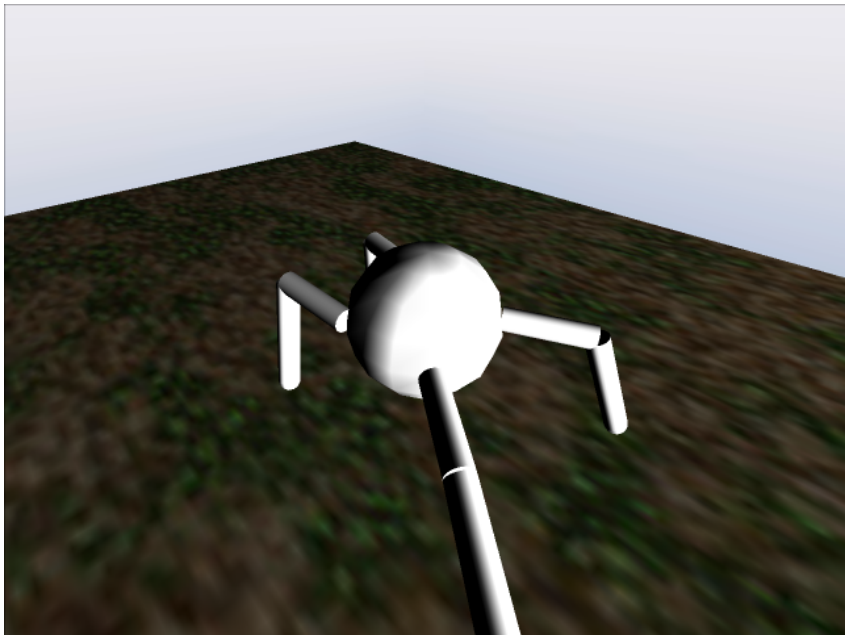


Fig. 8.2: The chrono_ant-v0

- **Minimize the energy consumption**
  The agent should not use more energy than necessary and avoid unnecessary movements.
- **Do not rest in joint limits**
  This point might be less evident. The excursions of the joints are limited (the sphere-leg in $\left[-\frac{\pi}{3},+\frac{\pi}{3}\right]$ while the leg-ankle in $\left[-\frac{\pi}{2},+\frac{\pi}{4}\right]$) and this might lead to policies that make the robot rest on the blocked joints to spare energy.

The reward is evaluated as:

$$r = v + a - 0.2\sum_{i=1}^{N} T_i\omega_i + 3\sum_{i=1}^{N} l_i \tag{8.1}$$

With:

r Timestep reward
v Average speed in the last timestep
a alive: 1 if alive, -1 if failed
i Sum over the number of actuators
N Total number of actuators
T_i i-th motor Torque
ω_i i-th motor angular speed
l_i i-th joint is at limit, =1 if True

8.3 Training and Results

8.3.1 Algorithm and Neural Network

To solve these environments we used the penalty version of Proximal Policy Optimization (PPO) 4.21 used to train a 3 Fully Connected Hidden Layers NN shown in 8.3, whose 1st hidden layer size is 10 times the number of inputs, the third layer size is 10 times the number of outputs and the second HL size is the geometric mean of the sizes of the other two.

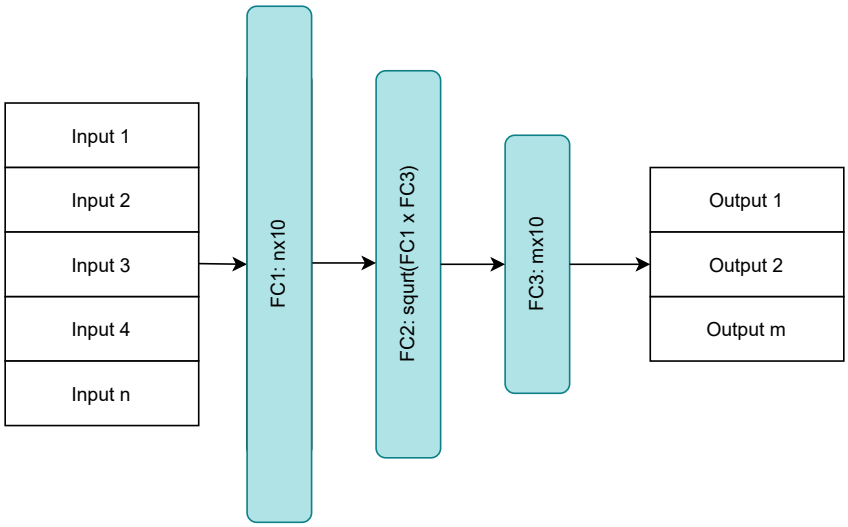


Fig. 8.3: NN architecture

8.3.2 Results

8.3.2.1 Ant Results

After 10000 episodes of training whose simulation was distributed over 20 CPU cores the agent learned to quickly walk straight without falling and touching the ground. It can be noticed that the distance travelled reaches the maximum as shown in figure 8.5, the sum of rewards slightly decreases in 8.4. This happens because we are rewarding the speed and the survival, this means that the

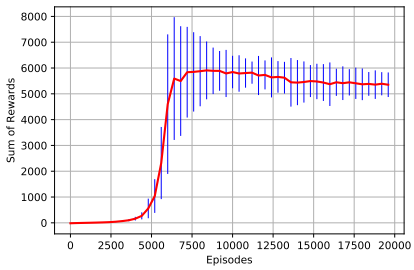


Fig. 8.4: Episode rew. sum (10 batches mean and var) [8]

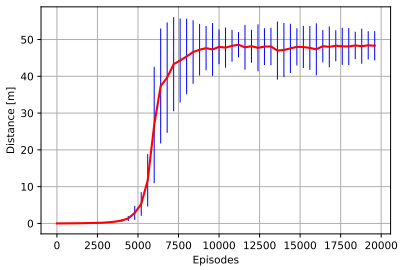


Fig. 8.5: Episode travelled dist. (10 batches mean and var) [8]

Euler Angles vs Quaternions

We investigated how the choice of quaternions instead of Euler Angles affects the policy learning process. Quaternions, unlike Euler angles do not suffer problems of singularity, the drawback is an additional input node and consequently more weights to optimize. As showed in fig. 8.6 and 8.7 the smaller NN taking Euler Angles as inputs learns faster as expected. It must be considered though, that while the advantage of this approach is limited, it can be undertaken only far from EulerAngles singularity conditions. For these reason, this approach is advisable only in particular cases after due consideration.

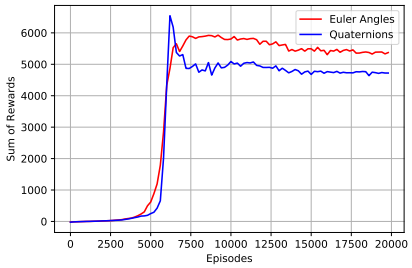


Fig. 8.6: Episode rew. sum (10 batches mean) [8]

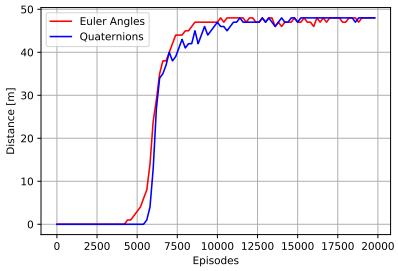


Fig. 8.7: Episode travelled dist. (10 batches mean) [8]

Input Scaling

As suggested by LeCun [26], backpropagating the error is more efficient when observation are centered and scaled. The case of legged walker makes no exception, as showed in fig 8.8 and 8.9. The number of episodes needed to converge doubles and the distance reached is shorter in the unscaled case.

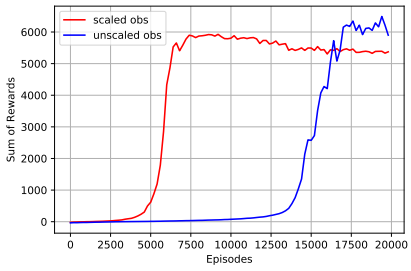


Fig. 8.8: Episode rew. sum (10 batches mean) [8]

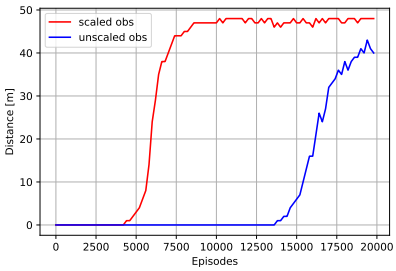


Fig. 8.9: Episode travelled dist. (10 batches mean) [8]

8.3.2.2 Partial Conclusions

Reaching this preliminary results demonstrated the potential of PyChrono as physics engine for RL environment given its capability of fast simulation of Multi Body constrained dynamics with non-smooth contacts. In addition, these first positive results with DRL techniques showed the synergy between DRL and Multi Body simulation, encouraging us to further develop PyChrono providing features that were not present by then, such as Anaconda deployment and several submodules.

Chapter 9
Virtual Environments for Neural Network training

Here we show how the 3D Cad plug-in of Project Chrono can been used to efficiently import models and we apply it to import a 6DOF robotic arm, create a DRL environment from it and train an agent it to solve a task. After, we present the gym-chrono project, a Python module that extends the well known OpenAI *gym* package with a set of physics based environment for continuous control.

9.1 From CAD 3D models to virtual environments

A major feature of PyChrono is the capability of imported mechanical models created from a 3D CAD tool, Solidworks®, by means of a plug-in. SolidWorks allows to enhance its GUI with new functionalities by adding libraries that are interfaced with SolidWorks thanks to its C# API.

From the 3D CAD model we are able to import bodies, centers of gravity positions, links, masses, inertia matrices and shapes. The plugin creates a Python script in which Solidworks 3D model are translated into PyChrono systems. Actuators have to be added manually from the Python PyChrono API.

9.1.0.1 Mass properties

Almost any multibody simulator is able to evaluate mass, COG and inertia matrix knowing the body shape and density by integrating numerically over the volume. This approach obviously assumes uniform density, thus being prone to errors in mass properties whenever the body density is highly non uniform. In a robotic link, for example, where the mass is concentrated in the motor. This kind of simulation-reality gap could prevent controls developed in simulation to work in real world.

CAD software allows to precisely evaluate mass properties of subsystem, considering the right density and shape of each component, and thus the simulation importing these data will be closer to reality.

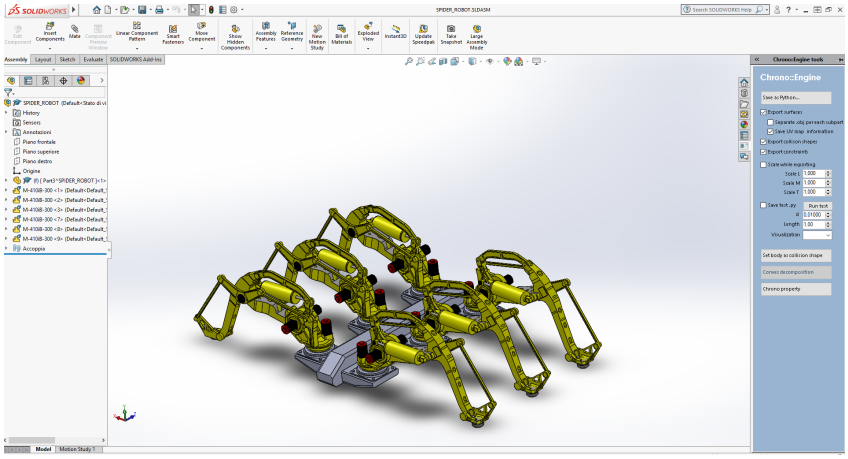


Fig. 9.1: The SolidWorks GUI with the Chrono plug-in

Table 9.1: Solidworks/Chrono system names comparison

Solidworks	Chrono
Part	Body
Rigid Subassembly	Body
Flexible Subassembly	Bodies and Constraints
Body	X
Coll. Body	Coll. Shape
Mate	Constraint (ChLinkMate)
Features	Triangular Mesh (Vis. Shapes)

Collision shapes are added through the plug-in that can set a Solidworks body (see table 9.1) as a collision shapes (automatically recognizing spheres, cylinders and boxes). The body is renamed to be recognized by the parser (such as COLL.Cylinder in figure 9.2 and its density is set as close to 0 to not interfere with the mass property.

This tool offers a convenient pipeline to generate PyChrono models, automating the process makes it faster and less prone to error. In addition, modifications to the 3D CAD model can be extended to the Multibody model just by launching the plug-in exporter.

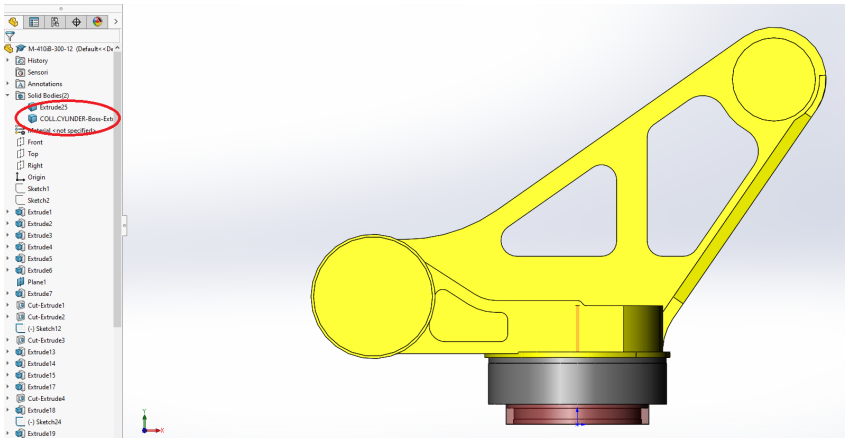


Fig. 9.2: The Chrono-SolidWorks plug-in collision shape addition process

9.2 Training to 6-DOF robotic arms to reach a goal and avoid collisions

9.2.1 Environment and Goal

The environments is composed of a 6 DOF robotic arm (Comau Racer3) and a small cube both placed on a flat floor. The robot is equipped with a 2-fingers grippers whose center has to reach the center of the cube, whose position is random within a working area. The agent can control the torque applied to the 6 rotational constraints. In the process the agent has to avoid:

- Contact between parts of the robot
- Contact between the gripper and the floor
- Blocked joints (at limit)
- Energy waste

In order to do so the reward is evaluated as follows

$$r = \frac{d_c}{||d|| + \varepsilon} + C_r + C_f + e_c||\omega \mathbf{T}|| + J_l \tag{9.1}$$

Where

d_c Distance reward coefficient
d Gripper-Target distance
ε To avoid division by 0
C_r Contact between robot parts. -1000 if there is self-collision, 0 otherwise.
C_f Gripper finger contact penalty, equals to the contact force on the
e_c Energy cost

ω Motor rotation speeds
$\mathbf{T}$ Motor torques
J_l Number of joints at limit

We treated differently the collision between robotic parts and between finger and ground: the fisrt one is to avoid at all cost so it is hugely penalized and terminates the episode, while the latter is only penalized proportionally to the contact force and does not abort the episode.

9.2.2 Modelling

After importing the SolidWorks model, only actuators have to be added, and searching for the concentric mates by name w can use their reference frames to define the motors. The process can be automated, indeed, once the model was set up for the Racer3 robot, there was no need to further modify the code when changing the robotic arm model (except for speed and rotation limits and maximum torques).

Contact Shapes

We added a limited number of contact primitive shapes (cylinders and boxes), the red shapes in figure 9.3. The plug-in has the capability to decompose complex shapes as convex hulls, but we wanted to investigate how a few contact primitives can lead to a working policy without sacrificing simulation speed.

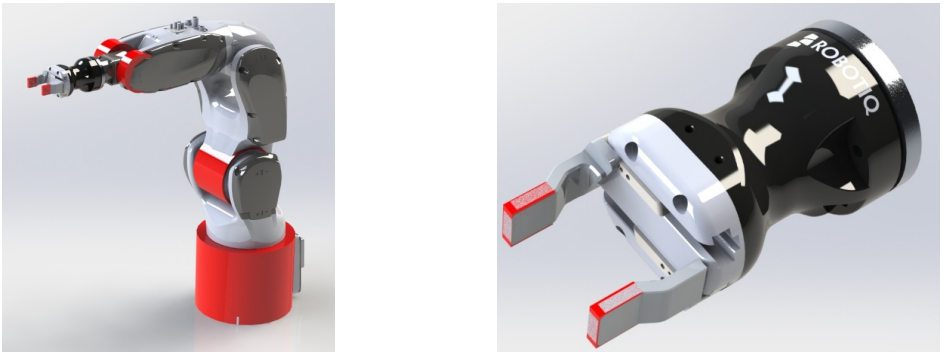


Fig. 9.3: Comau R3 collision shapes and gripper collision shapes [7] .

9.2.3 Training

The first approach was to fix the position of the box to speed up the training and then make it random and train further the NN. This proved to be unsuccessful, since the optimizer zeroes the weights of the state elements associated with the target position, being constant. When switching to random target position, the control always bring the gripper in the old fixed position, as in the reward progression in figure 9.4 . On the other hand, while randomizing the target position from the beginning makes the training longer and less stable, this approach can effectively train the NN to solve the task, as shown in the right plot in figure 9.4. From the simulation frame in figure 9.5 we can notice that the agent can place the gripper in the right position without touching the floor.

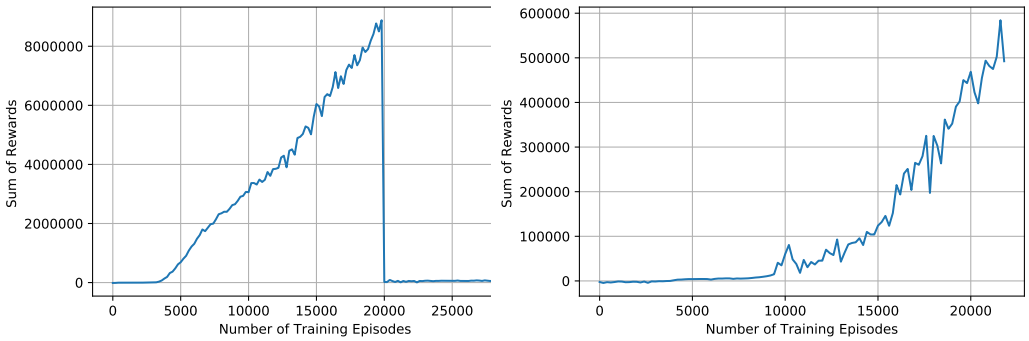


Fig. 9.4: Training approaches: from fixed target position to random target position vs random target position from scratch. [7]

9.2.4 Changing the Robotic Arm CAD Model

The usage of the plugin, together with the aforementioned strategy to find links by name to add limits and use their reference frame to add motors allowed us to switch to a different robotic arm (an ABB IRB 120) with ease. Given the 3D CAD model of the new robot that is exported by the plugin, the only modifications to the code are:

- The strings of parts and mates names
- Numerical data (maximum motor torque, maximum link rotations)

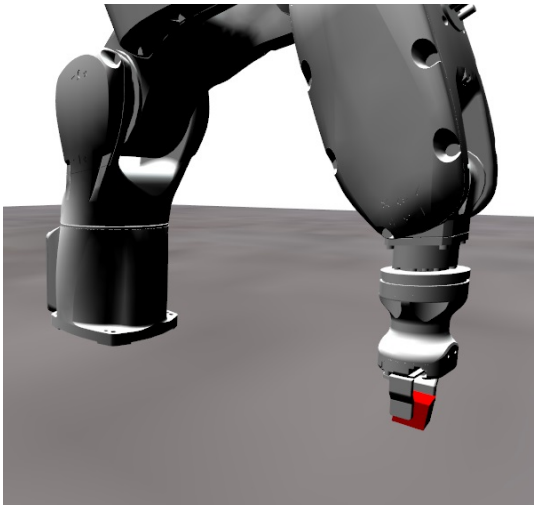


Fig. 9.5: Comau R3 Reaching Target Position [7]

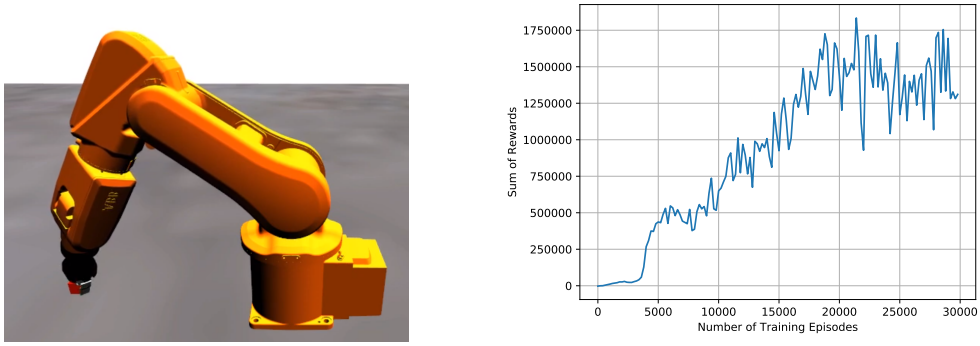


Fig. 9.6: A simulation screen showing the trained policy on the ABB and its reward progression [7].

9.2.5 Conclusions

After these experiments, we can state that:

- The Solidworks plug-in offers a valuable tool to create simulation models that can be used as DRL training environments
- A limited number of simplified collision shapes is enought for the training to address the policy in the right direction
- DRL techniques show good adaptivity to randomness in robotic tasks, as long as the randomness happens in the training as well

- The modularity of our platform makes easy to make changes to the robot or change it completely.

9.3 The gym-chrono project

9.3.1 Aim and Features

The OpenAI gym [12] python package allows to be extended through packages that provide additional environments. Providing environments compatible with OpenAI gym environments brings has two major upsides:

1. In the last few years many DRL framework have been developed, the most known being OpenAI Baselines, Stable-Baselines [22] and coach [13]. These are Python packages that provide several DRL algorithm and NN architectures to be used with gym-compatible environments. In general, the vast majority of algorithms available in GitHub expects a gym-compatible environment that has become a standard.
2. There are several tools, most of them coming from the aforementioned frameworks, that can be leveraged by gym-compatible environment. One of the most known and useful is SubprocVecEnv (Sub-process Vectorized Environments) that is a class that wrap gym environments to allow automatic exploitation of parallelization through multiprocessing.

In order to do so the package extending gym must provide each new environment in the form of a Python class the satisfies the following requirements:11

1. Inherit the gym.Env class.
2. Observation and Action of each space must be defined as instances of gym.Spaces classes.
3. Each environment is expected to override a set of method: reset, step, get_ob, is_done, ScreenCapture, Render, convert_observation_to_space, _set_observation_space, __setstate__ amd __getstate__
4. Environments must be registered and assigned to a specific is, such as chrono_ant-v0

The package is installed through pip:

```
git clone https://github.com/Benatti1991/gym-chrono
cd gym-chrono
pip install -e .
```

Once gym-chrono is installed, its environment can be trained using any RL framework and it works out-of-the-box. The following command launches OpenAI baselines to solve the Chrono Ant environment using PPO:

```
"python -m baselines.run --alg=ppo2 --env=gym_chrono.envs:chrono_ant-v0
```

```
--network=mlp --num_timesteps=2e7 --ent_coef=0.1 --num_hidden=32
    --num_layers=3 --value_network=copy"
```

In addition, gym-chrono can be a mold to create new environments using the infrastructure already in place to ease the process, while leveraging the API of PyChrono, the vehicle JSON file parser or the SolidWorks plug-in to generate models.

9.3.2 Environments

These are the environments provided by gym-chrono. We call the state s and the action a

9.3.2.1 Benchmark Environments

These environments are supposed to be solved in a reasonable amount of time and are used to benchmark algorithms or tune the training hyperparameters, without the aim of being models of real robots.

- **chrono_pendulum-v0**
  A reversed pendulum that is to be balanced. $s \in \mathbb{R}^4$, $a \in \mathbb{R}$
- **chrono_ant-v0**
  A 4-legged walker that has to run straight as fast as possible without falling. $s \in \mathbb{R}^{30}$, $a \in \mathbb{R}^8$

9.3.2.2 Robotic Environments

These environments are more complex and are based on real robots, whose mechanical properties and visual shapes were exported from CAD model using Chrono plug-in for SolidWorks.

- **ChronoRacer3Reach-v0**
  A model of a Comau Racer3 robotic arm, the agent has to bring the center of the gripper to the center of a randomly positioned target box avoiding collision with the floor or between parts of the robot. $s \in \mathbb{R}^{18}$, $a \in \mathbb{R}^6$
- **chrono_hexapod-v0**
  A 6-legged walker that has to run straight as fast as possible without falling. Each legs has 3 actuated joints so there are 18 motors to control. The robot is modelled after a real PhantomX Hexapod Mark II. $s \in \mathbb{R}^{53}$, $a \in \mathbb{R}^{18}$

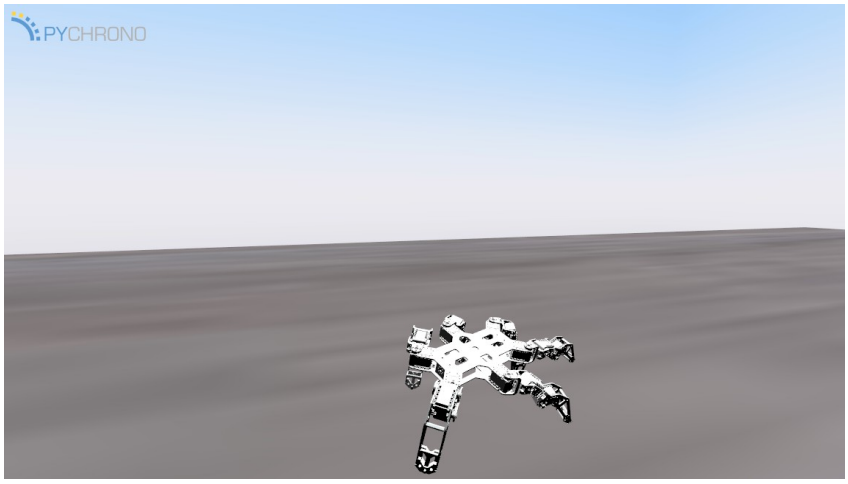


Fig. 9.7: gym-chrono Hexapod

9.3.2.3 Autonomous Vehicles Environments

This set of environments makes use of the Vehicle and Sensor module to create environments in which the agent has to learn to drive vehicles from simulated sensor data. [1]

These end-to-end autonomous driving tasks provided by gym-chrono can be divided into 2 sub-families:

1. **Vision-only observation**
   The observation consists of an RGB image provided by the camera sensor. These tasks require only CNN that performs 2D convolution on RGB image inputs so the convolutional architectures and the algorithms provided by any DRL framework can be used to solve them, but at the same time the image-only input limits the number and complexity of the applications.
2. **Tuple observation**
   A tuple is a set of heterogeneous objects. In DRL the observation might be composed of several tensors of different shapes, such as RGB images (3D tensors) depth buffers (2D tensors) and vectors containing the numeric outputs of several sensors; such as GPS, speedometers etc. (1D tensors). These environments require to treat the state as a tuple, and the NN should be tailored based on the tuple composition. For example, if the observation tuple contains an RGB image and a data vector, the first should be processed in a convolutional layer, while the second in a feed forward layer. It is self evident that given the vast variety of possible tuple observation structures (considering also that order matters) DRL

[1] These environments are the latest addiction and at the time of writing this document they haven't been added to the main gym-chrono repository. For this reason the environments that will be available in the main gym-chrono may be slightly different.

frameworks don't support tuple observation. For this reason we decided to implement a custom PPO algorithm 7. This being said, the capability of dealing with these environments allowed us to use DRL-techniques to solve more meaningful tasks, in which the perception of surroundings through cameras and sensor data (objective position, speed etc) are fed together to the policy to solve autonomous navigation tasks.

These environments being:

- **camera_obstacle_avoidance-v0**
  Vision-only observation. A car has to drive along a straight road while avoiding 3 randomly placed pillars. The agent can control the steer and throttle of the car. $s \in \mathbb{N}^{80\times45\times3}$, $a \in \mathbb{R}^2$
- **rccar_hallway-v0**
  Vision-only observation. A RC car has to drive along a path delimited by red cones on the left and green cones on the right. The agent can control the steer and throttle of the car. $s \in \mathbb{N}^{80\times45\times3}$, $a \in \mathbb{R}^2$
- **off_road-v0**
  Tuple-observation environment.The vehicle has to navigate in an off-road scenario disseminated with rocks to reach a goal position. Both starting and target position are randomly placed. Besides the RGB camera image, the agent also knows the relative position of the target, the direction of the target, its own heading and speed. $s \in \mathbb{N}^{80\times45\times3} \times \mathbb{R}^5$, $a \in \mathbb{R}^2$

Consider that every time we deal with RGB images, the domain of the image is only a subset of $\mathbb{N}$, since their value is in the $[0,255]$ (8-bit integer) interval.

Chapter 10
Deep Reinforcement Learning for Autonomous Vehicles

The utilities provided by Vehicle module of PyChrono allows to create detailed physical models for wheeled and tracked vehicles and, together with sensor simulation, gave us the ingredients to create environments for end-to-end DRL control techniques for AV.

End-to-end means that the control takes the sensor data as input and directly outputs the control values, in the case of AV the steering angle, the throttle intensity and the braking intensity.

We solved increasingly challenging tasks, going from unrealistic vision-only environments to autonomous navigation in an off-road scenario, the most meaningful of which are described in this chapter.

10.1 Vision-only tasks

10.1.1 Obstacle avoidance: camera_obstacle_avoidance-v0

10.1.1.1 Environment

An M998 High Mobility Multipurpose Wheeled Vehicle (HMMWV) has to drive on a straight road while avoiding 3 pillars whose position along the road width is randomized, as represented in figure 10.1. The agent takes as input an RGB image from a sensor camera on the hood of the car and controls the steering and the throttling.

The reward is the car progress speed (advancement along the road), on which is applied a bonus of 3000 if the car makes it to the end. The episodes is aborted if the car touches a pillar or falls off.

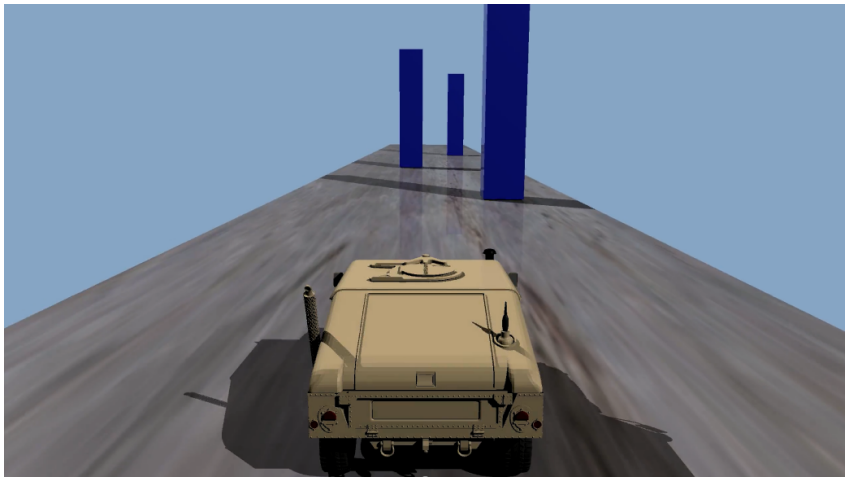


Fig. 10.1: Obstacle avoidance Environment 3D person view

Control and Simulation frequencies

Viable values for the timestep in our wheeled vehicle simulations are between 1 and 3 ms. Keeping the same timestep for the simulation and the control would be both computationally unfeasible and unrealistic, since the amount of data to be passed and memorize if controlling at that frequency would be impossible to manage and would severely slow down the simulation, moreover autonomous cars control frequencies are much lower. For this reason we set simulation timestep and control frequency independently. The step method of the environment has an internal loop that advances the physics and the visualization until the control timestep is reached.

Prescribing feasible actions

We use a continuous action algorithm to solve continuous autonomous driving control task, meaning that the controller can prescribe any control value in the range. For example, the steering is $\in [-1,1]$, where -1 is all left and 1 is all right. This being said, the prescribed value might not be reachable in the current situation, in other words there is a limit upon the maximum rate of change of a controlled value, at least in reality. To respect this, we clamp the prescribed action in a feasible action range, such that the agent either performs that action or at least try to get as close as it can to it.

This strategy has been applied to every AV environment.

10.1.1.2 Algorithm and Results

The PPO algorithm implemented in 7.2 was used to train the CNN as in 7.4.0.2. In figure 10.2 is a scheme of the input, the NN and the output.

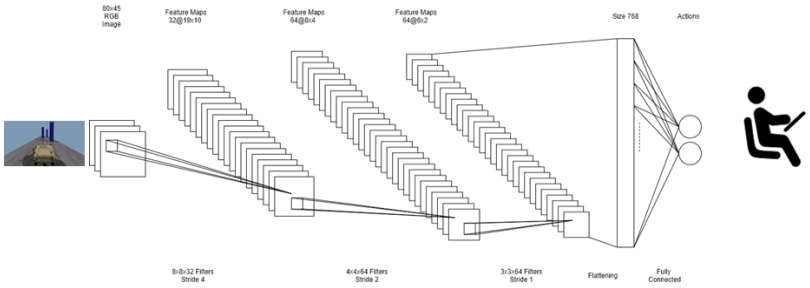


Fig. 10.2: Obstacle avoidance Reward Progression

The reward progression (figure 10.3) shows how after 220 episodes the mean reward is over 3000, meaning that the end of the road is always reached. This environment combines:

- A relatively easy task
- Minimal graphics complexity: the only triangular mesh is the HMMWV chassis, and it is not used during training
- The relatively simplified model: simplest and faster tire model on rigid terrain, minimal suspension model.
- The low camera resolution (80x45): fewer weights to train, smaller training set (so smaller GPU memory footprint). [1]

For these reasons it can be trained successfully on commodity hardware in a reasonable amount of time.

10.1.2 Lane driving in realistic scenario: rccar_hallway-v0

This is another meaningful vision-only task environment was created and solved and is reported here because it features a realistic indoor navigation environment,

[1] The image size affects the memory footprint of the training process since the states sampled must be kept in memory to train the model. The images are preponderant in terms of memory compared to the other quantities memorized (actions, rewards, VF, log probabilities). The dataset is collected step by step on the RAM memory and passed to the GPU memory so actually it does not affect the GPU memory only but GPU memory is typically smaller and more likely to be depleted.

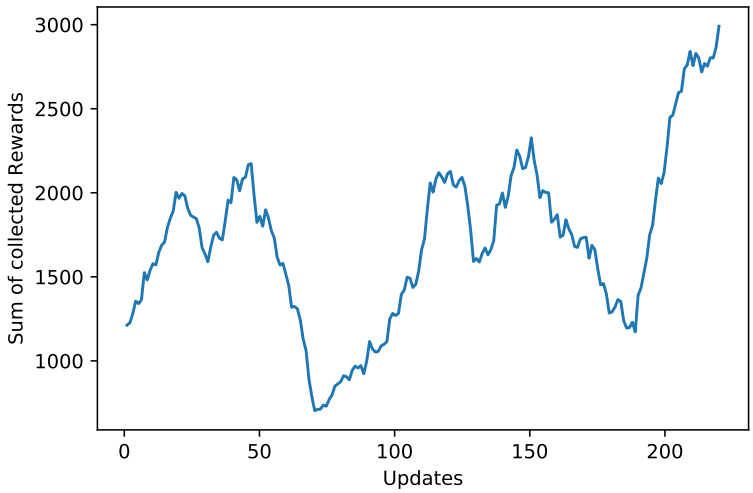


Fig. 10.3: Obstacle avoidance Reward Progression

as shown in figure 10.4. The agent control a small RC car [2] to drive inside a track delimited by cones.

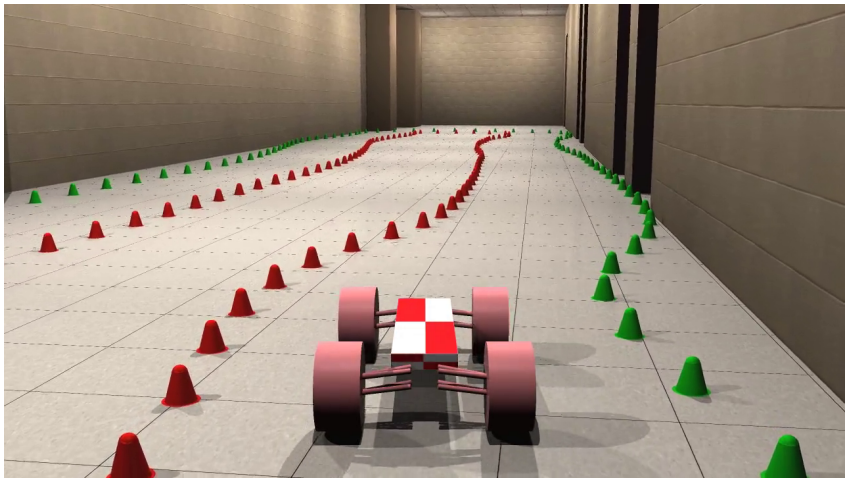


Fig. 10.4: Hallway RC Car environment 3D person view

[2] This policy was meant to be deployed on the real RC car equipped with an Nvidia Jetson owned by the SBEL Lab of UW Madison, but the COVID-19 forced us to change our plans.

The NN takes a 160x90 image as inputs and has only 1 output: the steering, since the throttle is controlled by a feedback controller that keeps the speed constant.

The training algorithm and NN are identical to the ones used for the obstacle avoidance, except the different number of actions and input image size. The larger memory footprint of the environment (a large mesh for the scenario and numerous objects) together with the higher number of updates required to converge makes this environment more challenging than the previous one. We also noticed that placing the starting point on a straight prevents the convergence of the policy, since the starting null steering angle is a local maximum for the policy. On the other hand placing the starting point on a turn makes the agent solve the task as shown by the reward progression in figure 10.5.

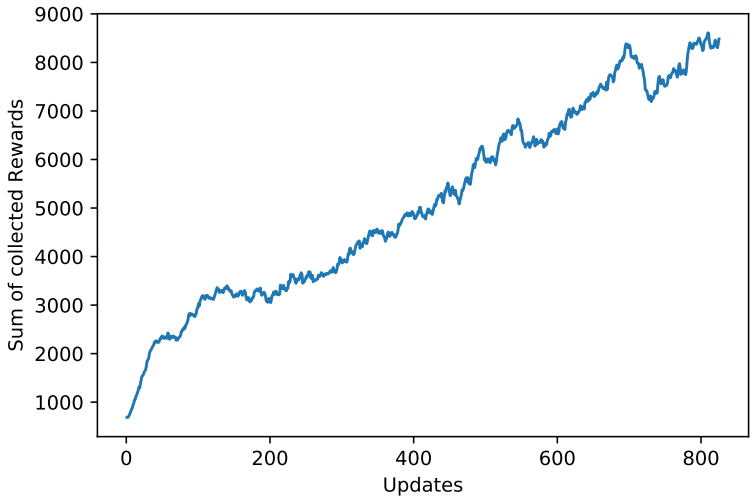


Fig. 10.5: Hallway RC Car Reward progression

10.2 Multi-sensor tasks

The results obtained in vision-only end-to-end control tasks, although encouraging, have little potential for real-world applications. In fact, the camera observation cannot carry direction information thus it is limited to applications in which the path is made obvious, such as in the cone-delimited track. In realistic on-road and off-road scenarios the car needs additional information in order to effectively navigate, such as its position, heading, the position of the target to reach and other meaningful data (speed, tangential acceleration etc.) while still needing camera data to be aware of

the surroundings. For these reasons we enhanced the PPO algorithm to manage tuple observations and added a NN able to take an image and a vector as input. [3] These tools allowed the data from several sensors (camera, GPS, IMU, Speedometer) to be fed to the agent achieving sensor fusion, thus being able to solve more complex navigation tasks.

10.2.1 Moving vehicle convoy

In this environments a HMMWV has to follow a convoy of 3 HMMWV while avoiding obstacles placed near the path. We call the car controlled by the agent the *Follower*, while the vehicle it has to follow (the last of the 3) is called *Leader*.

The agent receives an image from the camera sensor together with a 5-element vector, being composed of: the GPS coordinate difference w.r.t. the leader, the heading of the vehicle, the speed of the follower and the speed of the leader.

The action controlled are the steering angle, the throttle and the brake. We condensed throttle and brake into 1 single action, where 1 means maximum throttle and -1 maximum braking. This was done to avoid simultaneous braking and throttling while reducing the complication of the control by lowering the number of outputs.

Performance Considerations

Training an agent to control a vehicle to follow a convoy makes necessary to have a vehicle, preferably more than one, to follow during training. Simulating 2 vehicles (a Leader and a Follower) would slow down the simulation and the training process with it, let alone simulating additional vehicles in the convoy. At the same time, it is crucial to put the leader vehicle in the training, otherwise the agent cannot be trained to tell it apart from an obstacle. For this reason we adopted what we call *Ghost Leaders*, which are visualization-only vehicles composed of a single body (with the full visualization mesh attached) whose position and orientation are imposed by the path to follow. At each timestep the Ghosts advance along the prescribed path and are rotated such that their axis is tangential to the path. Visually they look like 3 vehicles driving close, except for the lack of wheel rotation and oscillation on the suspensions, but these visual details do not affect the training.

[3] To solve multi-sensor environments while still relying on third-party DRL frameworks some append an extra channel to an RGB image and populate it with the vector input. In other words, to process a WxHx3 3D tensor and a N 1D tensor together, they append an additional layer to the 3D tensor (that becomes WxHx4) and populate the first N elements of its first row with the vector. This approach has major drawbacks though: it would waste a lot of memory since only a small portion of the additional size of the 3D tensor would be used, plus while the image is an array of 8-bit integers, the vector is an array on double precision floating point numbers. Considering the limitations of this approach, we decided to avoid it and implement our algorithm capable of dealing with multiple sensor data.

Path randomization

The path is defined by a Bezier curve, along which the ghost leaders are advanced at each timestep, and 8 obstacles, randomly picked from a set of rocks and trees, are placed right next to the path. The paths feature severe turns, and the speed increases linearly such that the agents is trained to follow the convoy in different and difficult conditions. In addition, to avoid overfitting any particular configuration or path, we randomly flip about the X axis and rotate about the Z. The rotation is picked randomly from a set of four angles set: 0°, 90°,180° and 270°. This means that the possible paths are 16 (2x2x4) as shown in figure 10.7.

Reward

The reward evaluation is based on the relative position of the follower w.r.t. the leader. The reward value depends on the length $|d|$ and the angle α of the distance vector. In other words, we reward the follower being close and in a "cone" whose point is on the leader. In addition, being closer than the optimal distance does not increase the reward.

Algorithm 13: Rew evaluation

if $\alpha \in \left[\frac{5}{4}\pi, \frac{7}{4}\pi\right]$ **then**
  $r = \frac{C_d}{max(|d|-d_o,0)+\varepsilon}$
else
  r=0
end

Where:

α Is the angle of the distance in the leader reference
d_o Is the optimal distance
C_d Coefficient (to tune this reward with failure penalties)
ε To avoid reward explosion when $|d| - d_o \sim 0$

10.2.1.1 Algorithm, Progression and Testing

The algorithm in 7.2 has been used with the tuple option, coupled with the tuple observation NN architecture as in 7.4.0.3. More on this will be explained in 10.2.2, that comprehends and expands the same subject.

The NN has been trained from scratch for 1600 updates, each updates being happening each 300 timesteps on 5 parallel episodes. To increase the speed we used a simplified model of the vehicle, switching to the full model only after convergence was reached.

The obtained policy has undergone thorough testing in which we tested the robustness of the policy in several conditions, some of which different and more chal-

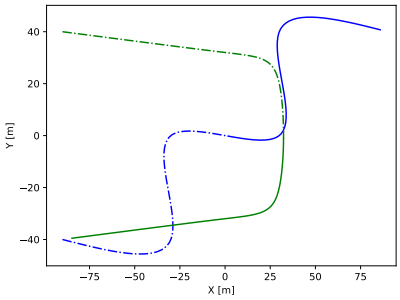


Fig. 10.6: C and double S ”base” paths.

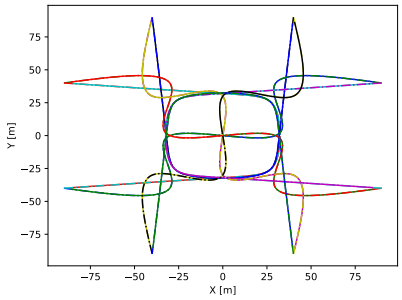
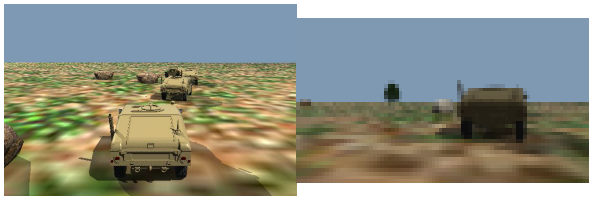


Fig. 10.7: All the 16 possible paths for he convoy.



(a) (b)

Fig. 10.8: Convoy following third-view camera vs camera sensor observation 80x45 pixels

lenging respect to the training. In particular we tested convoys composed of a single leader and multiple simulated follower (in testing the leader is simulated too) on both rigid and deformable terrain, as shown in figure 10.8.

Moreover, we used a path generator to generate paths able to reach the opposite corner of the map while avoiding obstacles randomly placed and scaled in size. To enforce non trivial paths 4 large obstacle are placed along the line between the starting position and the goal, and many more of various size are randomly distributed on the rest of the map. The variety of paths is represented in figure 10.11.

To simulate 4 vehicles driving on a deformable terrain we used SynChrono, a module of Chrono that leverages MPI parallelism to interface several ranks,a rank being a simulation of a physical system running on a node of a cluster computer. In this case there are 4 ranks each one of them responsible of simulating a single vehicle. This infrastructure is not available from the Python API, and introducing deformable terrain would slow down the training anyway. For these reason these kind of realism was only introduced in testing.

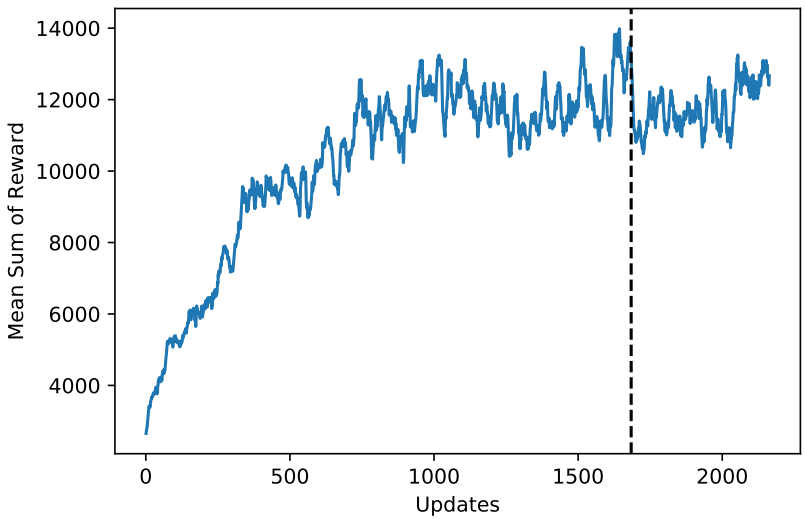


Fig. 10.9: Plot of the moving average of the sum of collected rewards with respect to the policy updates. The vertical dashed line represents the switch from the reduced to the full HMMWV model.

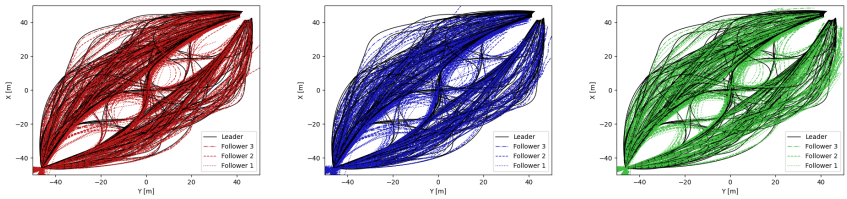
(a) (b) (c)

Fig. 10.10: Still frames from attached third person camera: (a) rigid terrain; (b) SCM-Hard terrain; (c) SCM-Soft terrain taken from testing scenarios

Results Discussion

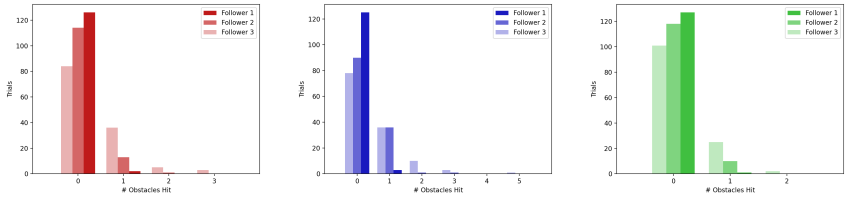
From the results obtained running statistical analysis on rigid, deformable-hard and deformable-soft terrain we can state that:

- The policy obtained training on 1 follower can be generalized to a higher number of autonomous followers and keep good performance, as shown by histograms in figure 10.12. We observe that the number of obstacles hit does grow behind in the convoy, but the difference is rather small.



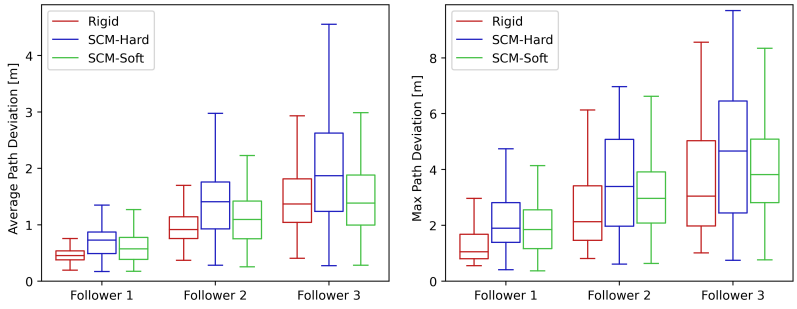
(a) (b) (c)

Fig. 10.11: Paths of every vehicle on each terrain type: (a) Rigid; (b) SCM-Hard; (c) SCM-Soft.



(a) (b) (c)

Fig. 10.12: Number of obstacles hit by each vehicle on the three terrain types: (a) Rigid; (b) SCM-Hard; (c) SCM-Soft.



(a) (b)

Fig. 10.13: Statistics of average and maximum follower path deviations.

- The deviation from the leaders path grows along the convoy, and this is clearly visible in figure 10.13, where the position deviation is much larger for the last follower. This happens because the follower tends to cut the turn, and it is also the main cause of the higer number of hits.
- The loss in maneuverability introduced by hard deformable terrain has a negative impact in both path deviation and numbers of hits, nut the damping effect of soft SCM balance it to the point that in soft SCM tests we consistently get better results.

We can conclude that we can effectively train a NN to control a car to follow a convoy in an off-road scenario. This being said, DRL techniques might not be the ideal way to get a tightly packed convoy, since the agents should just do whatever the leader does without any need for AI. For these reasons, the upcoming work on convoys will be oriented towards widespread formations, where the follower has to keep a certain relative position w.r.t. the leader, thus being impossible to imitate the same actions without hitting an obstacle

10.2.2 Off-road navigation with obstacles

In this environment an autonomous off-road vehicle (a John Deere *Gator*, as in figure 10.14) has to navigate through obstacles and reach a target position from the information coming from camera, IMU, GPS, and speedometer sensors and knowing the GPS coordinates of the goal in a 80x80 meters terrain map.

In addition, we wanted the policy to be able to navigate on hilly terrains.

Reward

We reward the rate of change of the approaching of the vehicle towards the target:

$$\frac{d_{old} - d_{new}}{\delta t}$$

Moreover, we penalize and abort whenever the vehicle hits an obstacle, goes off the map or reaches the maximum time allowed (20s). On the other end, the episode is terminated with a huge reward bonus (+2500) when the vehicle reaches the goal).

The agent can control the steer and throttle/brake of the vehicle (just as before) and the task might seem easier compared to the convoy following, but there are 2 caveat:

- In the convoy the follower always starts behind the leaders, so the agent starts in a good position, and it just have to stay close and behind the convoy to succeed. In this case the vehicle starts from an arbitrary position and rotation, and the right course of actions is less evident than before.

Fig. 10.14: Picture of a John Deere Gator from John Deere Italian website

- This task requires longer term planning: in terms of goal approaching driving around an obstacle might mot be optimal in the short term since it deviates from the straight line direction, but deviating from the shortest path is the only way to reach the goal and get the maximum reward.

Randomization

The position of the goal and the initial position of the Gator are randomized by placing the vehicle on the radius of a circumference whose diameter is equal to the terrain size, and the goal is placed on the opposite side of the circle. In polar coordinates, given the random angles $\theta \in [0, 2\pi] \quad \alpha \in [-\pi/2, \pi/2]$ and the terrain size $S = 80m$, the initial position of the vehicle is S, θ and the position of the goal is $S, \theta + \alpha$.

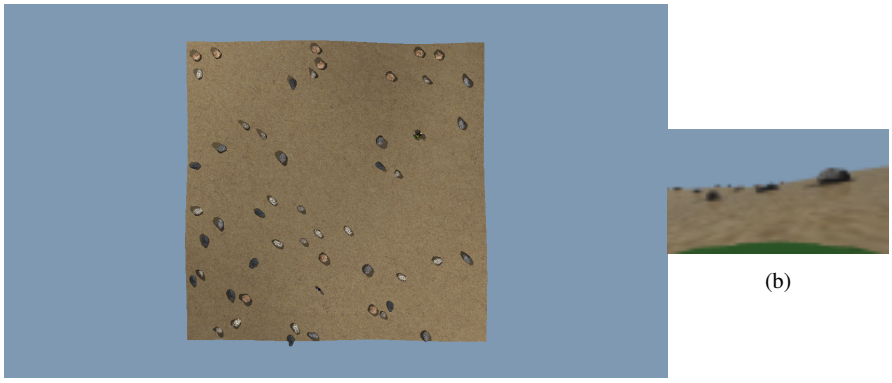
The algorithm placing the obstacles uniformly distributes them on the terrain, while avoiding the immediate proximity of the initial position of the vehicle and the goal position.

Both the obstacles and the terrain texture are picked randomly from a set.

10.2.2.1 Algorithm, Learning Technique, NN architecture

Algorithm and NN architecture

The algorithm is identical to the one in 10.2.1.1, and so is the NN. This being said the NN architecture will be explained in this section since this task enlightened



(a) (b)

Fig. 10.15: Bird view of the navigation environment and camera sensor observation 80x45 pixels

the performance difference between different architecture. In particular, a different architecture that processed the two inputs separately and had only 1 hidden layer between the fusion and the output, still performed well on convoy following, while not being able to solve this task.

This happens because in the former task the camera observation was predominant, while in this case the camera and the position observations are both equally important, and in most cases the action should consider both weighing up the information available, to avoid the obstacle while approaching the target.

This happens because there is an hidden layer between both the vector input and the flattening of the CNN, and 3 hidden layers between the concatenation and the output. This allows to weight the inputs before the fusion and then perform complex processing on them, as the 3 HL allow.

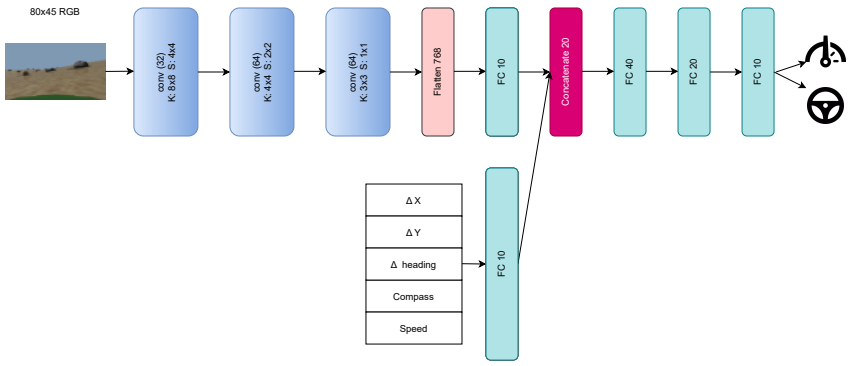


Fig. 10.16: Actor neural network architecture

Local Coordinates

In principle it is easy to evaluate the relative position of the goal in the vehicle reference frame given the absolute position of both and the heading angle of the vehicle, and it might seem that a NN should not have problems to be trained to infer this relation. In practice, this is not what happens, as demonstrated in the plots in figure 10.17, where the task is not solved even with no obstacles involved. This might be due to the heading angle discontinuity near 0 and 2π.

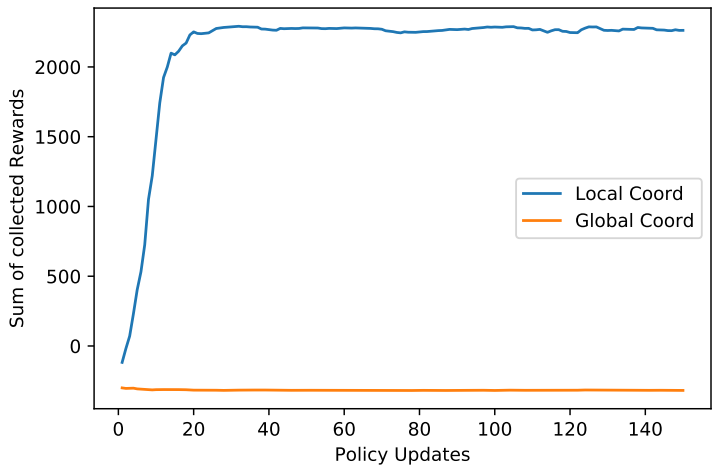


Fig. 10.17: On flat terrain ***without obstacles***, the algorithm converges immediately when feeding the distance from the goal rotated in the vehicle reference frame. When the distance is given in the global frame, training does not converge.

Considerations on Minibatch size

As previously mentioned, this task requires some degree of planning. DRL is capable of taking actions that pay off in the long term since it maximizes the expectation of future rewards. At the same time, though, the estimate of the gradient in the ADAM algorithm 7 might be inaccurate, leading to unstable progression. For this reason using larger minibatches sampled for larger datasets greatly stabilized the training. We ended up feeding 1000 elements MB sampled from training stets of 6000 transitions.

10.2.2.2 Results

Curriculum Learning

We adopted a curriculum learning approach [9], progressively increasing the complexity of the task, as shown in Fig. 10.18. Int the first part of the training the terrain is flat with a random number of obstacles (from 0 to 30). After $\sim$ 200 policy updates the confergence is reached thus the obstacle number is increased to be always 30, and after a drop the reward is capped again. When, after 376 policy updates, the flat terrain is replaced with hilly terrain (while keeping the same number of obstacles), the agent struggles and many updates are required in order to converge again. In the third and last stage of the training, the obstacle count was increased to 50 and the terrain texture was randomized. Curriculum has proved to be necessary since convergence could not be directly reached from scratch on hilly terrain. We found that:

- Irregularities in terrain height cause policies trained exclusively on flat terrains to perform poorly
- This problem can by solved by undergoing further training on hilly terrain
- The curriculum learning approach (see Fig. 10.18) was crucial in handling the complex tasks; i.e., hills with many random obstacles.

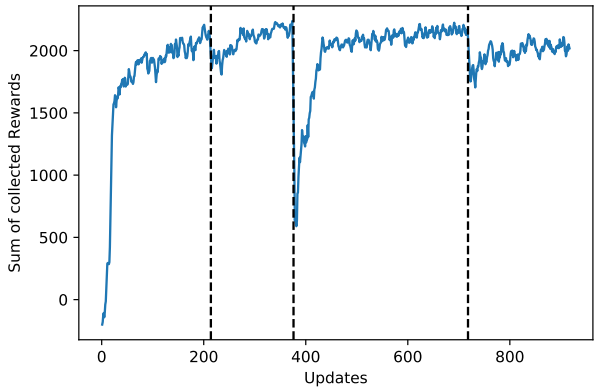
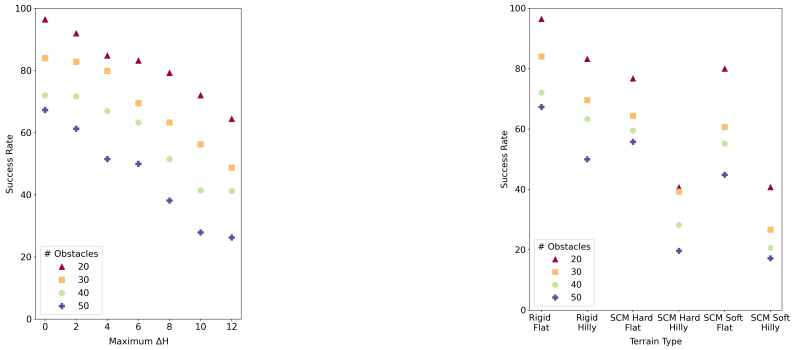


Fig. 10.18: Reward progression. Vertical lines represent the changes introduced to make the environment more challenging. In order of occurrence, the dotted lines mark: fix obstacle count at 30; change from flat to hilly terrain; and increase of obstacle count to 50.

Statistical Analysis

We tested our policy running 200 simulations per each configuration, increasing the number of obstacles, scaling the obstacles randomly, varying the severity of the terrain irregularity and trying different terrain textures.

As easily predictable, increasing the number of obstacles, the height of the hills and the softness of terrain all play a role in reducing the success rate. This being said, the success rate is low only when these 3 effects are combined.



(a) Success rate in % vs hilliness on rigid terrain.

(b) Success rate in % vs hilly and flat terrains with varying stiffness.

Fig. 10.19: Success rate when increasing complexity of the terrain.

Conclusion

The training and testing of these policies allows us to state that the synergy between end-to-end DRL techniques and our simulation platform has interesting capabilities in solving complex navigation tasks in partially observable environments. In addition, the high but adjustable level of simulation detail showed us the importance of full vehicle simulation and how it can be used to refine policies trained on simpler physical models.

Chapter 11
Conclusions

This work made evident the synergies and reciprocal benefits between DRL and simulation, and in the process we reached several goals:

- We developed a physics simulation Python module, PyChrono, that binds to Python the majority of Chrono classes. This module provides rigid and flexible body constrained dynamics with both smooth or non-smooth contacts, vehicle simulation, 3D rendering, vehicle simulation and advanced ray tracing based sensor simulation. This module offers superior dynamics simulation capabilities compared to simplified vehicle models provided by the most known driving simulation used for autonomous driving.
  The features provided, together with deploying it as an Anaconda package, made PyChrono a largely used simulation tool, and it has been used by researchers in companies and universities [28], to the point that the package has been downloaded more than 3000 times and PyChrono has been mentioned in a Nature paper [19] about Python computing.
- We created multiple DRL environments, and organized them in a Python module that can be installed trough Python package manager to extend OpenAI gym with DRL environments based on PyChrono. The module provides simpler benchmarking environments, robotics tasks and autonomous driving environments featuring vehicle and sensor simulation. These environments are fully compatible with both algorithms and utilities developed for OpenAI gym, such as OpenAI Baselines.
- We implemented a state-of-the-art DRL algorithm (PPO) capable of dealing with tuples of heterogeneous input tensors and various deep NN architectures to be used in different situations.
- We solved increasingly challenging control tasks. The last presented in particular, being a vehicle navigating in an outdoor environment that reaches a goal while avoiding obstacles based on GPS and camera sensor data, has no precedent in literature.

References

1. Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
2. Mihai Anitescu. Optimization-based simulation of nonsmooth rigid multibody dynamics. *Math. Program.*, 105(1):113–143, 2006.
3. Mihai Anitescu and Gary D. Hart. A constraint-stabilized time-stepping approach for rigid multibody dynamics with joints, contact and friction. *International Journal for Numerical Methods in Engineering*, 60(14):2335–2371, 2004.
4. Mihai Anitescu and Alessandro Tasora. An iterative approach for cone complementarity problems for nonsmooth dynamics. *Computational Optimization and Applications*, 47(2):207–235, 2010.
5. David M. Beazley. Swig: An easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, TCLTK'96, page 15, USA, 1996. USENIX Association.
6. R. Bellman, Rand Corporation, and Karreman Mathematics Research Collection. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957.
7. Simone Benatti, Alessandro Tasora, Dario Fusai, and Dario Mangoni. A modular simulation platform for training robots via deep reinforcement learning and multibody dynamics. In *Proceedings of the 2019 3rd International Conference on Automation, Control and Robots*, ICACR 2019, page 7–11, New York, NY, USA, 2019. Association for Computing Machinery.
8. Simone Benatti, Alessandro Tasora, and Dario Mangoni. Training a four legged robot via deep reinforcement learning and multibody simulation. In *Multibody Dynamics 2019. ECCOMAS 2019. Computational Methods in Applied Sciences*, 2019.
9. Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proc. 26th Intern. Conf. on Machine Learning*, pages 41–48. ACM, 2009.
10. Christopher Berner, G. Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, D. Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, R. Józefowicz, Scott Gray, C. Olsson, Jakub W. Pachocki, M. Petrov, Henrique Pond'e de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, J. Schneider, S. Sidor, Ilya Sutskever, Jie Tang, F. Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019.
11. Rogerio Bonatti, Ratnesh Madaan, Vibhav Vineet, Sebastian Scherer, and Ashish Kapoor. Learning visuomotor policies for aerial navigation using cross-modal representations, 2019.
12. Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.
13. Itai Caspi, Gal Leibovich, Gal Novik, and Shadi Endrawis. Reinforcement learning coach, December 2017.
14. François Chollet et al. Keras. https://keras.io, 2015.
15. Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2019.
16. Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.
17. Christopher Goodin, Matthew Doude, Christopher Hudson, and Daniel Carruth. Enabling off-road autonomous navigation-simulation of lidar in dense vegetation. *Electronics*, 7(9):154, 2018.
18. Shixiang Gu, Ethan Holly, Timothy P. Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation. *CoRR*, abs/1610.00633, 2016.

19. Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, and et al. Array programming with numpy. *Nature*, 585(7825):357–362, Sep 2020.
20. Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S. M. Ali Eslami, Martin A. Riedmiller, and David Silver. Emergence of locomotion behaviours in rich environments. *CoRR*, abs/1707.02286, 2017.
21. Toby Heyn, Mihai Anitescu, Alessandro Tasora, and Dan Negrut. Using krylov subspace and spectral methods for solving complementarity problems in many-body contact dynamics simulation. *International Journal for Numerical Methods in Engineering*, 95(7):541–561, 2013.
22. Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.
23. K. Hornik. Some new results on neural network approximation. *Neural Networks*, 6(8):1069 – 1072, 1993.
24. Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *IN PROC. 19TH INTERNATIONAL CONFERENCE ON MACHINE LEARNING*, pages 267–274, 2002.
25. Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.
26. Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. *Efficient BackProp*, pages 9–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
27. Timothy Lillicrap, Jonathan Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, 09 2015.
28. Pasindu Lugoda, Leonardo A. Garcia-Garcia, Sebastien Richoz, Niko Munzenrieder, and Daniel Roggen. Shapesense3d: Textile-sensing and reconstruction of body geometries. In *Adjunct Proceedings of the 2019 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2019 ACM International Symposium on Wearable Computers*, UbiComp/ISWC '19 Adjunct, page 133–136, New York, NY, USA, 2019. Association for Computing Machinery.
29. J. Madsen, N. Pechdimaljian, and D. Negrut. Penalty versus complementarity-based frictional contact of rigid bodies: A CPU time comparison. Technical Report TR-2007-05, Simulation-Based Engineering Lab, University of Wisconsin, Madison, 2007.
30. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.
31. OpenAI, :, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation, 2018.
32. Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.
33. Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.
34. John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.
35. John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2015.

36. John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.
37. Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pages 621–635. Springer, 2018.
38. David E. Stewart. Convergence of a time-stepping scheme for rigid body dynamics and resolution of Painleve's problems. *Archive Rational Mechanics and Analysis*, 145(3):215–260, 1998.
39. David E. Stewart. Rigid-body dynamics with friction and impact. *SIAM Review*, 42(1):3–39, 2000.
40. David E. Stewart. Reformulations of measure differential inclusions and their closed graph property. *Journal of Differential Equations*, 175:108–129, 2001.
41. David E. Stewart and Jeffrey C. Trinkle. An implicit time-stepping scheme for rigid-body dynamics with inelastic collisions and Coulomb friction. *International Journal for Numerical Methods in Engineering*, 39:2673–2691, 1996.
42. D.E. Stewart. Existence of solutions to rigid body dynamics and the Painlevé paradoxes. *C. R. Acad. Sci. Paris*, 325:689–693, 1997.
43. Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.
44. Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *CoRR*, abs/1804.10332, 2018.
45. Alessandro Tasora and Mihai Anitescu. A matrix-free cone complementarity approach for solving large-scale, nonsmooth, rigid body dynamics. *Computer Methods in Applied Mechanics and Engineering*, 200(5-8):439 – 453, 2011.
46. Alessandro Tasora, Radu Serban, Hammad Mazhar, Arman Pazouki, Daniel Melanz, Jonathan Fleischmann, Michael Taylor, Hiroyuki Sugiyama, and Dan Negrut. Chrono: An open source multi-physics dynamics engine. In *HPCSE*, 2015.
47. Joshua Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. *CoRR*, abs/1703.06907, 2017.
48. E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.
49. Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.
50. Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.
51. David Wolpert and William Macready. No free lunch theorems for search. 03 1996.
52. Huaiqin Wu. Global stability analysis of a general class of discontinuous neural networks with linear growth activation functions. *Inf. Sci.*, 179:3432–3441, 09 2009.
53. Yuke Zhu, Ziyu Wang, Josh Merel, Andrei Rusu, Tom Erez, Serkan Cabi, Saran Tunyasuvunakool, János Kramár, Raia Hadsell, Nando de Freitas, and Nicolas Heess. Reinforcement and imitation learning for diverse visuomotor skills, 2018.