IMPLEMENTATION AND COMPARATIVE ANALYSIS OF MACHINE LEARNING METHODS FOR THE CLOSED-LOOP CONTROL OF FLUID FLOWS

LORENZO SCHENA

S265722

Master of Science in Aerospace Engineering - Aerogasdynamic

Politecnico di Torino - von Karman Institute for Fluid Dynamics

Supervisors:

Politecnico di Torino
prof. S. Pieraccini

von Karman Institute for Fluid Dynamics
prof. M. A. Mendez
F. Pino (PhD)

Apr 2021

To my family and friends, for their unabated support during these years.

ABSTRACT

The steep ascent of machine learning techniques has also had an impact on fluid mechanics in the past few years. In the light of its great achievements in solving complex problems, machine learning-based techniques seem to be a very promising solution to address flow control problems. Due to its inherent nonlinearities, non-convexity, and high-dimensionality, it offers a challenge for machine learning methods that learn by trial and error, such as the cutting-edge technique known as Deep Reinforcement Learning (DRL). This work aims at studying the performances of such an approach in three test cases: the 1D linear advection equation, the 1D Burgers equation, and the control of a von Kàrmàn Vortex Street behind a 2D cylinder at Re=100 (J. Rabaud et al in [24]). The performance of two machine learning techniques, DRL and Bayesian Optimisation (BO), was assessed. Moreover, a thorough hyperparameters optimisation campaign is carried out, to find the best tuning for the RL algorithm at stake. Finally, a benchmark will be carried out, to gain an overview of how machine learning approaches relate to other optimisation approaches for optimal closed-loop control.

ACKNOWLEDGEMENTS

This Master Thesis project fits inside a broader research topic developed at the Von Karman Institute (VKI) by my advisor, F. Pino, and my supervisor, prof. M. A. Mendez. I would like to thank both for their assistance and their guidance during this Master Thesis project. Their ingenuity towards ideas inspired me deeply.

I would also extend my sincere thanks to my supervisor at Politecnico di Torino, prof. S. Pieraccini, without whom this beatiful experience at VKI would not have been possible, for her support.

Lastly, I should also thank M. Desmet for his help with the code.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PYTHON CODES

LIST OF ALGORITHMS

ACRONYMS

ML	Machine Learning
RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
HPO	Hyper Parameters Optimisation
BO	Bayesian Optimisation
PSO	Particle Swarm Optimisation

1 INTRODUCTION

1.1 THE GRAND CHALLENGE IN CLOSED LOOP CONTROL

Manipulating the dynamics of a flow is a major interest in fluid mechanics. Controlling a fluid flow may lead to advantages that span from suppressing or promoting instabilities to reduce drag and improve efficiency. This research direction led to different achievements, for example: aerodynamic drag reduction, lift increase or a minimisation of the environmental impact optimising combustion or reducing noise pollution, just to name a few. Moreover, active flow control is critical to off-design conditions in dealing with flow systems.
In order to design a control strategy, a simulation is needed most of the time to model the flow behaviour with and without control. Flow control problems are described by non-linear and high-dimensional partial differential equations. In particular, to face the inherent high dimensionality of the control problem applied to fluid flows, classic approaches propose a variety of tools that rely on different kinds of projection of the Navier-Stokes equations into some reduced basis - a process called *reduced order modelling (ROM)*.

This reduced model is finally used as a foundation of the controller design. As it could be expected, different controllers lead to different control performance and robustness - i.e.: the sensitivity to reduced order model perturbation.

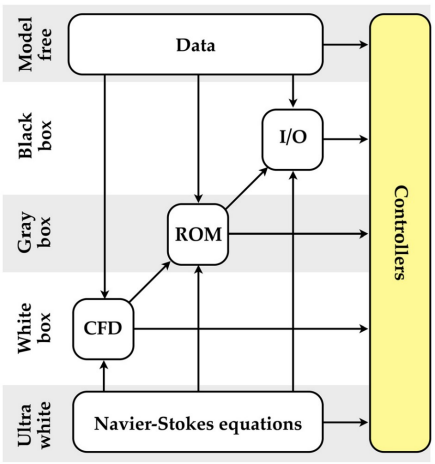

Figure 1: Approaches to Control from Brunton et al [4]

However, besides this reduced order model approach - also called *gray box* - several other approaches can be used: ranging from using Navier-Stokes equation to derive the control law analytically - i.e.: *white box* - or considering inputs/outputs relationship only, referred as *black box methods* since they are opaque to the underlying physics of the system at stake. From a theoretical point of view, also directly dealing with the Navier-Stokes is possible, an approach often called *ultra-white box*.

However, directly solving those equations in order to apply some control in real time is still a prohibitive task from a computational standpoint and it is seldom used if not in some simple test case.

Recently, thanks to the great availability of data, increasing computational power and the wide spreading of Machine Learning and optimisation techniques the idea of directly *learning* the control-law relying on input/output data only - hence following the *black-box approach* - is becoming more and more popular. Furthermore, this black-box approach requires little if any tuning in changing the application area. This is a change of paradigm with respect to classical control design methods that rely on expertise in the modelling of the process at stake and that generally are strictly linked to the operational design condition.

This data-driven control problem then may be stated as a constrained optimisation problem where the objective is to discover a mapping, from input of the system to its output, $y \rightarrow u$ that satisfies a given cost-functional $J(s,u)$ where s is the performance measurement - for example the information received by a sensor - and u is the control input. Namely, using measurements in order to evaluate and correct the control law is the very definition of *closed-loop feedback control*.

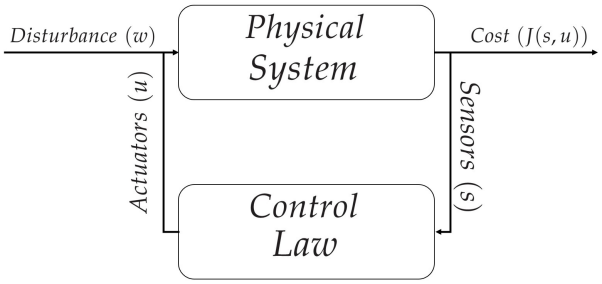

Figure 2: Closed-Loop Feedback Control scheme

1.2 FRAMEWORK OF THIS PROJECT

At the von Karman Institute for Fluid Dynamics, my supervisor prof. Mendez and my advisor F. Pino (PhD) are applying this *data-driven approach* - among many others - in an industrial framework: the control of an instability called undulation that appears in the jet-wiping galvanisation process in certain operating conditions.
The *galvanisation* is an industrial process of coating iron and/or steel with Zinc (Zn). When exposed to air, the pure Zinc react with the Oxygen (O_2) creating a layer of *zinc oxyde* (ZnO) which protects the material underneath from corrosion. One of the most extensively used technique *hot-dip galvanisation*.

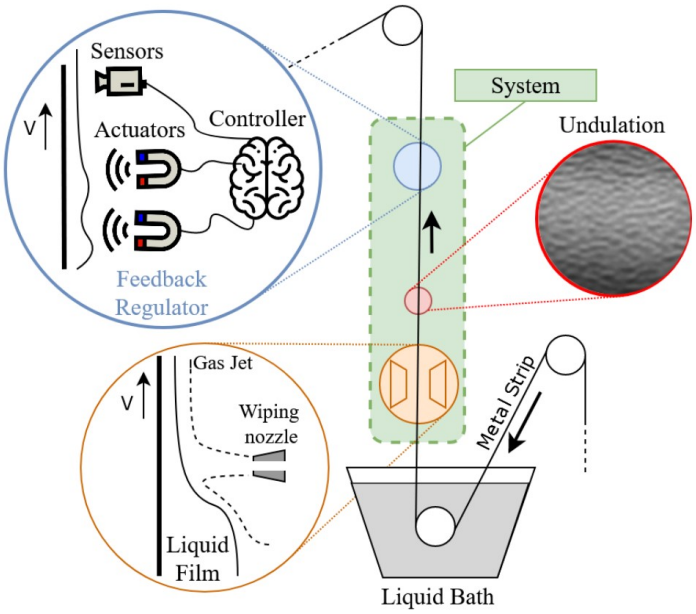

Figure 3: Project scheme

As can be seen by the scheme in Fig. 3, this technique consists in immersing a metal plate into a bath of molten zinc, withdrawing it and then some impinging gas actuators allows to control the final coating thickness. However, some instabilities occur in the process: the liquid film trigger jet oscillations that combined with inertia engender the formation of undulation patterns in the final zinc coating. Due to that drawback, the operational condition are limited.

With such a complex physics to be handled, the design of a controller following the black box approach, hence without the burden of detailed modelling is of utmost interest. For this reason, in this work *black-box* approaches are investigated, in order to preliminary assess if such a mindset would allow to obtain satisfying performances while simultaneously avoiding the burden of setting up a complex mathematical formulation for the derivation of the control law.

1.3 SCOPE OF THIS WORK

This thesis rises inside the broader research topic illustrated above and aims to contribute to it by addressing different questions:

i) What are the outcomes when Reinforcement Learning is applied to a control problem of interest?

At first, a benchmark on a defined set of test cases will be performed employing *Reinforcement Learning* techniques. Such a method aims to learn directly the control law without any intermediate model

identification step - a powerful change in the whole control theory paradigm.

ii) Can the performances of Reinforcement Learning on such a task be improved? If yes, how?

Secondly, having collected the results on the test cases, some way to improve the performances will be searched. Two approaches will be followed:

- Improve the Reinforcement Learning model definition;
- Tailor *hyper parameters* of Reinforcement Learning method on the problem at stake - a process referred to *Hyper Parameters Optimisation (HPO)*.

iii) Reinforcement Learning methods implicitly deal with non linearities. How far is this results from a naive linear approach?

Being the performance review (on the test cases of interest) completed, a study of the control laws learnt in this way will be carried out. Moreover, a comparison between those and a straightforward *linear approach* will be performed.

1.4 THESIS OUTLINE

This thesis is organised as follows: in Chapter 2 a general overview on the main methodologies that will be tested during this work will be given: at first, the *Reinforcement Learning* problem will be posed. After that, the *Hyper Parameters Optimisation* framework will be analysed, investigating how these optimisation techniques could enhance the performance of Machine Learning methods and how they could also be a valid alternative to those altogether. After, the control laws obtained in such a way will be studied trying to learn from those. In Chapter 3 the test cases will be presented, including their implementations. Then, Chapter 4 presents an overview of the Results obtained. Finally, Chapter 5 includes a discussion of such results.

2 METHODOLOGIES

2.1 FROM FEEDBACK CONTROL TO REINFORCEMENT LEARNING

Dealing with high-dimensional and non linear phenomena, diving deeply in the formal modelling of a dynamical system can be time and computationally heavy. A valid alternative to the former approach can be offered by *model-free* control methods where the physics of the system itself can be learnt from the output that such a system produces instead of relying on some beforehand modelling. This wide category of *data-driven* control methodologies include, among many others, *Machine Learning Control* (MLC). Moreover, these controllers, often named "smart" controllers since they require little input by the user, have the capability to be general-purpose as they can be applied to different problems with no substantial modification. Under the umbrella of MLC, one of the most promising state of the art method is *Reinforcement Learning* and its combination with Neural Networks: *Deep Reinforcement Learning*.

Reinforcement learning has been recently placed under the spotlight of the scientific community after the groundbreaking result of defeating the world greatest AlphaGo grand master achieved by DeepMind in 2016 [30]. Since then, Reinforcement Learning has played a major role in a broad range of control applications, from robotics (see, for example [13]) to Healthcare problems, such as closed-loop blood glucose control in [7].

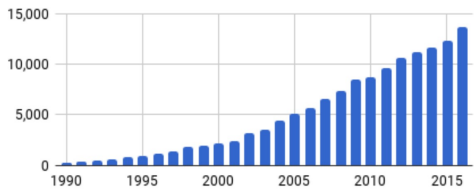

Figure 4: Growth of published reinforcement learning papers. Shown are the number of RL-related publications (y-axis) per year (x-axis) scraped from Google Scholar

Being fluid dynamics historically concerned by high dimensionality and non linearities, a great attention by the community was directed towards this state of the art technique. For instance, in the work by J. Rabaud et al. in [24], deep reinforcement learning is applied in order to achieve a reduction in the aerodynamic drag generated by the von Karman vortex street past a cylinder.

2.2 MACHINE LEARNING BACKGROUND

The general purpose of any ML technique is generally to extract information from data. This data-driven process can be carried out in different ways, each one with its pros, cons and field of applicability. MLC is composed mainly by three paradigms:

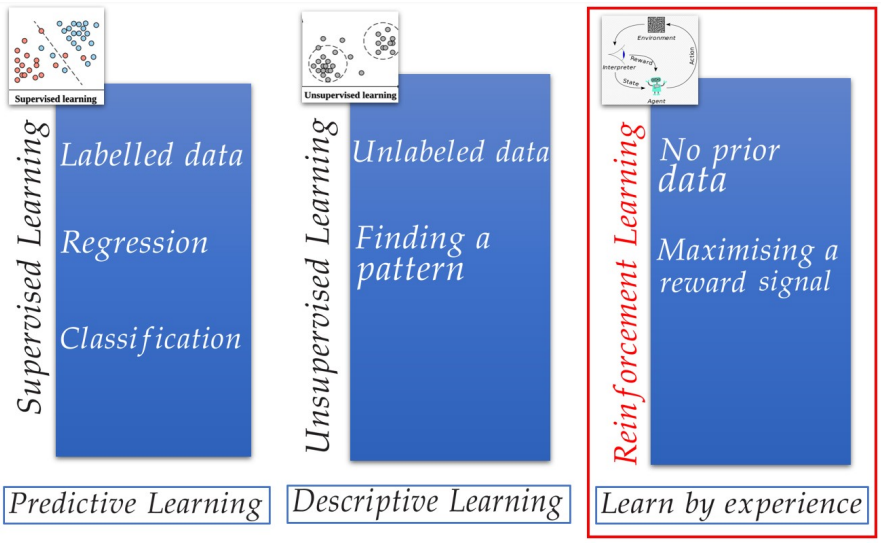

Figure 5: Machine Learning Categories

- **Supervised Learning:** the learning relies on data labelled by an expert, providing additional information to the algorithm. For example, this kind of ML could be used to compute a regression given a labelled data set such that it fits a mathematical criterion defined by the user.
- **Unsupervised Learning:** the data here is not labelled beforehand. Hence, the features have to be extracted directly from data specifying some criteria. Example of problems assessed with such an approach are dimensionality reduction and clustering, among others.
- **Reinforcement Learning (RL):** is a framework oriented to successive decision making ([31]) in order to maximise a reward signal defined before of the simulation. The data is not labelled nor given by an expert but the algorithm - or *agent* - generates such a dataset by interacting with the system of interest - or *environment*.

2.3 NEURAL NETWORKS

Neural networks are the core elements on which Deep Learning is built. As an high level definition it could be said that neural networks

are multi-layer networks of neurons that are used for a wide range of topics, from classifications to predictions. A neuron is a mathematical function that takes some inputs, weights them and finally sum them in order to produce some output. It receives some inputs through the *input layer* that are further elaborated throughout the *activation function* for a variable number of *hidden layers* and then, finally, an *output layer* closes the structure.

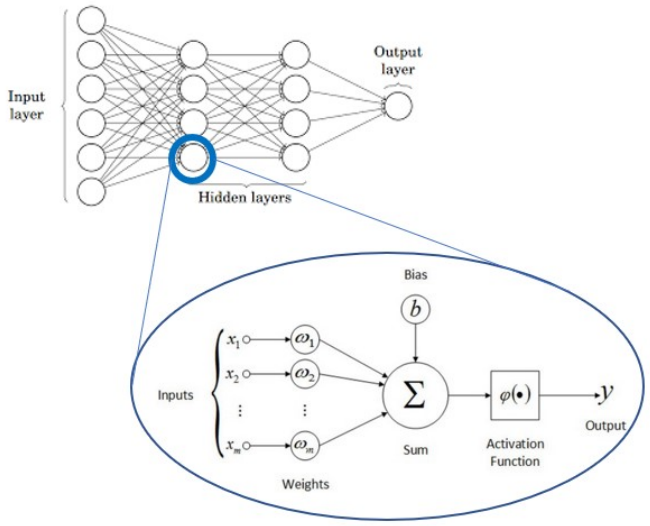

Figure 6: Neural networks overview

Neural networks are essentially black-box and non linear, universal functions approximators, where the "universal" adjective follows the theory of K. Hornik et al. in [9] which affirms that any function can be approximated by a neural network with the proper number of neurons, its central computational element.
Looking at the picture above, it can be stated that:

$$y_{i,j} = \Sigma_{i,j} f_{i,j}(\omega_{i,j} x_{i,j} + b_{i,j}) \tag{1}$$

where,

- $y_{i,j}$ is the output of the neuron i of layer j;
- $f_{i,j}$ is the activation function, a function that defines the output of a neuron given its input;
- $\omega_{i,j}$ is the weight of the i-th feature;
- $x_{i,j}$ the i-th feature;
- $b_{i,j}$ is the bias.

A *cost function* $J(\omega)$, function of the prediction (*out*) and the actual data (*out**), is introduced to define the optimisation problem, comparing *out* and *out**. During training, the gradient gives information

about the training direction - i.e.: on how to update the neurons parameters. Different architectures might be used, from which descend different properties. However, for the purpose of this work only two of them are briefly introduced:

I - Multi Layer Perceptron (MLP)
An MLP is a NN that contains more than one *layer*. Between the input layer and the output layer there are now some intermediate layers that are called *hidden layers*, because the computations carried out here are not visible to the user. A simple MLP of in Fig. 6 where all the neurons (or perceptrons) of one layer are connected to the following one is called *feed forward network* because the information is fed successively from layer to layer, from input to output.
It has to be noted that usually, when identifying the dimensionality of the network - i.e.: describing how many layers are present and how many neurons per each layer - the input layer is not included in the count since it does not make any computation.
The characteristics of the NN can be expressed in a mathematical form:

$$\bar{h}_1 = \Phi(\omega_1^T \bar{x}) \tag{2}$$

$$\bar{h}_{j+1} = \Phi(\omega_{j+1}^T \bar{h}_j) \; \forall \; j \; in \; \{1, ..., k-1\} \tag{3}$$

$$\bar{o} = \Phi(\omega_{k+1}^T \bar{h}_k) \tag{4}$$

Where Φ is the *activation function*, a function applied on the results of the internal computation of the neuron before they are effectively transmitted. The equations above represent the description of the input layer with the first hidden one, the internal connection of the hidden layers and finally the last hidden layer with the output layer.

II - Long Short-Term Memory Networks (LSTM)
Unlike the MLP, the LSTM architecture has feedback connections. LSTMs are suitable to process not only instantaneous data but also a sequence. This characteristic make them interesting when it comes to making predictions based on time series data even if a computational cost must be paid with respect to "classic" MLP.

However, instead of using this net as a standalone it could be combined with an MLP, assolving the function of a *feature extractor* - a way to transform the input data if the incoming information is too large or too noisy, helping the overall learning of the net.

2.4 REINFORCEMENT LEARNING (RL) BASICS DEFINITIONS

Reinforcement Learning aims to train an agent to make decisions. The agent receives the current state S_t and basing on that it makes

an action A_t interacting with the environment. As a consequence of such an interaction the environment evolves to a new state S_{t+1} and the agent receives a reward R_{t+1} for its behaviour.

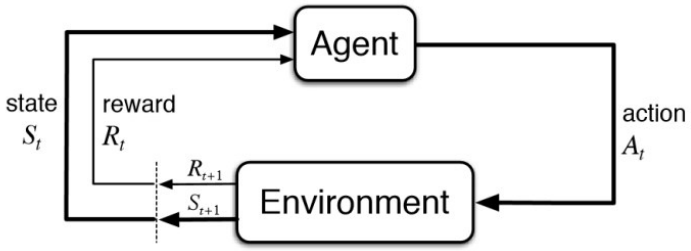

Figure 7: Reinforcement Learning steps

The action that the agent chooses starting from the current state S_t is encoded inside the *policy*.

Definition 2.4.1. A *Policy* π is a function which maps elements of the state space to elements of the actions space.

The policy fully defines the decision making process carried out by the agent. It could be *deterministic* or *stochastic* if the output is a parameter of a probability density function.

$$\pi : \mathcal{S} \to \mathcal{A} \textit{ deterministic} \qquad \pi : \mathcal{S} \times \mathcal{A} \to [0, 1] \textit{ stochastic}$$

Intuitively, it is desired that a trained agent will be making actions that have satisfactory results when interacting with *environment*.

Definition 2.4.2. The *Reward* (or reward signal) R_t is a number obtained by the agent for each action it takes. The objective of the RL agent is to maximise this reward on the whole simulation.

$$R_t = \sum_{t=0}^{\infty} \gamma r_t \quad \gamma \in [0, 1] \tag{5}$$

where γ is the *discount factor*. The discount ensures that immediate good rewards are preferred to future ones.

At first, since *there is no labelled data* available, the policy suggests random actions in order to explore the action space. This phase is called **exploration**. As a result of this phase, a preliminary dataset of *trajectories* (τ) is created. These are sequences of states, actions and rewards experienced by the agent in the environment:

$$\tau = (s_0, a_0, r_1, s_1, a_1, ..., r_T, s_T)$$

This dataset is passed to the neural networks inside the policy that will take the state as an input and output an action. This combination of RL with NNs is called **Deep Reinforcement Learning (DRL)**.

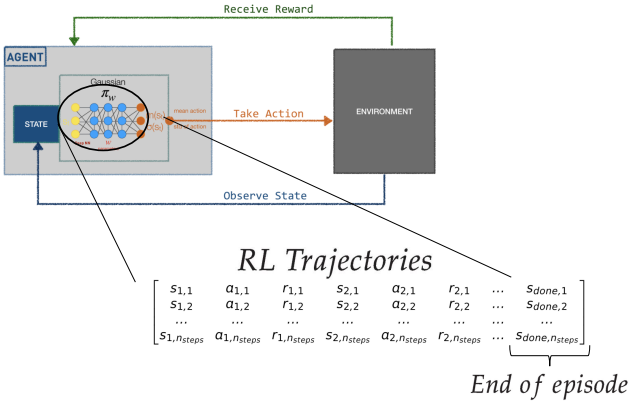

Figure 8: Deep Reinforcement Learning scheme

After having collected enough trajectories, the policy moves to an **exploitation** phase in which the existing trajectories will be refined in order to search for the highest reward. The *exploration - exploitation dilemma* - i.e.: when to stop exploring and start exploiting - is an active topic of debate in the ML community.

Automatically, if an action gets a low reward then the agent is not likely to make it again. This continuous process of learning how to behave inside the environment leads to an evolving policy across the simulation. How the policy is updated splits the learning method in two categories:

Definition 2.4.3. *On-Policy learning*: the policy improved during training is the same used for action selection.

In other words, in *on-policy methods* the agent learns "on the go" during the simulation - or game. This approach is computational effective but it is also dangerous: if the agent learns a bad policy it would be quite difficult for it to recovery. This issue is called *sub-optimal policy* learning. To partially face this problem, *trust region policies* have been developed. These will be discussed later on.

Definition 2.4.4. *Off-policy learning:* the policy improved during training is different from the one used for action selection.

It can be easily understood how the *reward shaping* highly influences the decision making process of the agent inside the environment: carefully designing the attention can be posed on some facets of the simulations or to others.

Hence, a reward is an *feedback* for the action taken inside the environment. On the other hand, another function keeps track of what is best on "the long run": the *value function*.

Definition 2.4.5. The *Value Function* V_π specifies the overall reward R that the agent can expect to obtain from a given state s, given a policy π.

$$V^\pi(s) = \mathbb{E}[R|s, \pi] \tag{6}$$

where $\mathbb{E}$ is the expectation operator.

The *expected value* of a discrete random variable is the proability-weighted average of all possible values. As an example, for a random variable X, it can take value x_1 with probability p_1, value x_2 with probability p_2, and so on, up to value x_N with probability p_N. Then the expectation of this random variable X is defined as:

$$\mathbb{E}[X] = \sum_{i=1}^{N} x_i p_i \tag{7}$$

Thus, the expected value is what one expects to happen on an average. The same holds for continuous random variables, except that the sum is replaced by an integral and the probabilities by probability density functions (PDFs). Let X be a continuous random variable with range $[a, b]$ and PDF $p(x)$. The epected value of X is defined by:

$$\mathbb{E}(X) = \int_a^b xp(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} x_i p(x_i) \tag{8}$$

An approximation of the above integral in a discrete sampled environment is given by the sum of sampled variables multiplied by their proability. The estimate of a function $f(x)$ where the indipendent varible x follows a probability distribution p is defined by:

$$\mathbb{E}_{x\sim p}(f(x)) = \int_a^b f(x)p(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)p(x_i) \tag{9}$$

As a consequence, actions that lead the agent into states of highest value would be preferred to highest reward. However, the value function is far more complex to evaluate because it has to be computed from the observations of the agent during the whole simulation, while the reward is obtained directly interacting with the environment.
A similar function is the *Action-Value Function*:

Definition 2.4.6. The *Action-Value Function* - also called Q function - is the expected return starting from state s, taking action a and from then on following policy π:
$Q : \mathcal{S} \times A \to \mathbb{R}$

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[R|S_t = s, A_t = a] \tag{10}$$

The relationship between Q^{π} and V^{π} is given by:

$$Q^{\pi}(s_t, a_t) = r(s_t, a_t) + \gamma V^{\pi}(s_{t+1}) \tag{11}$$

Finally, it is possible to define the *Advantage Function* which specifies the potential benefit of taking specific action a compared to other actions, from the given state s and following a policy π.

$$A(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \tag{12}$$

The last element needed to fully define a RL problem is the *environment*.

Definition 2.4.7. The environment is a mathematical modelling of the system that is going to interact with the RL agent. It is often possible to state the problem as a *Markov Decision Process* (MDP). This ensures that the state transition is affected by the previous state only and not from the transitions history - i.e.: each state satisfies the *Markov Property*.

Definition 2.4.8. The *Markov Property* is a characteristic property of the state. The states for which the transition to the next state of the environment is predictable without any need for information from the past states but only the present state. From a mathematical perspective, the necessary conditions for a state St to be defined as Markov is that the state transition probability function is exclusively conditioned by the present state:

$$\mathcal{P}[S_{t+1}|S_t] = \mathcal{P}[S_{t+1}|S_1, ..., S_t] \tag{13}$$

Definition 2.4.9. A *Markov Decision Process* is an environment in which all states satisfy the Markov property. An MDP is often defined as a tuple (S, A, $\mathcal{P}$,R,γ) where S is the state space, A is a finite set of actions, $\mathcal{P}$ is the state transition probability function, R is the reward function and γ the discount factor.

A method is called *model-based* if, instead of using only simulation samples to estimate rewards, some kind of cost function is encoded in the environment. Moreover, in such a case, possible actions and states are considered possible even if they are not experienced yet. This process is called *planning*. If otherwise such a model is not present a *model-free* method is defined. In such a case the learning process is based on trial-and-error solely.

2.5 IMPORTANCE OF RANDOM SEEDS IN MACHINE LEARNING APPLICATIONS

Stochastic algorithms rely on randomness. However, when a computer is asked to generate a sequence of random numbers they are *pseudo*random - i.e.: the result depends deterministically by the number (or vector) used to initialise such a process: *the seed*.

In the context of this work, that means that the behaviour of the RL agent is dependent on the random seed used to initialise the experiment - for instance, the initial weights of the neural networks. This is especially important when dealing with *on-policy* models which use the same policy for learning and for action sampling, as explained in Sec. 2.4. Hence, in order to achieve a meaningful portrait of the technique at hand, reporting only the best performance between the n seeds tried might lead to some misinterpretation. To assess this problem, in this work *four different seeds* have been used in each simulation, with the goal of obtaining more descriptive results.

Finally, it can be noted how the use of constant seeds certainly helps reproducibility of performances in different settings - for a given algorithm and task.

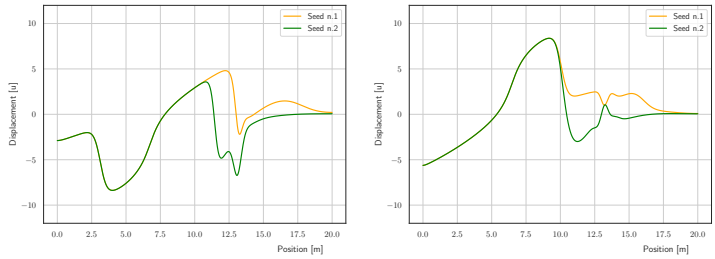

Figure 9: Importance of random seeds. In the pictures above, a given DRL controller tries to control a wave. All the parameters are the same: algorithm and environment are identical. The only difference is the initial seed. It can be noted how the evolution of the system is significantly different.

2.6 IMPLEMENTED ALGORITHMS

Recently, with an increasing interest of the community, illustrated by Fig. 4, a variety of Reinforcement Learning algorithms have been released open-source and it is therefore possible to use them in RL-based research. In this broad spectra of tools available, *Actor-Critic* methods stand out in terms of performance. To this broad category belongs state of the art algorithms like *Proximal Policy Optimisation (PPO)* and *Trust Region Policy Optimisation (TRPO)*, introduced in 2018 by J. Shulman et al in [28] and [27] respectively.

2.6.1 *Actor - Critic Methods*

The main idea is to split the model in two parts: one that computes an action based on a state and another one to compute the Q values (see Sect. 2.4.6) of the action. Following the convention in the papers and books, the first one will be called the *Actor* and the second one the *Critic*.

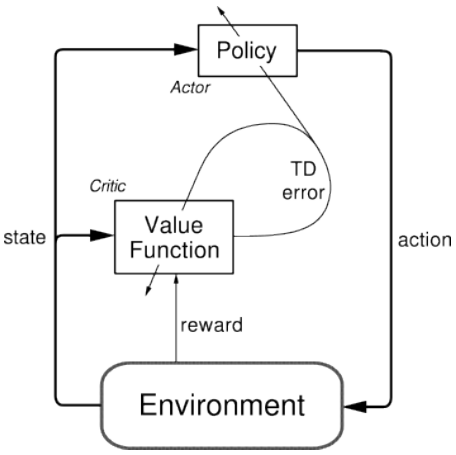

Figure 10: Actor-Critic Scheme from Sutton and Barto, 2018 [31]

Essentially the actor is responsible for the behaviour of the agent, dealing with the learning process of the policy (*policy - based*). On the other hand, the critic evaluates the value function (see Sect. 2.4.5) of the action (*value based*). It is expected that both get better with experience.

The actor can be a function approximator like a NNs (see Fig. 6) that takes as input the given state and outputs an action. The critic can be a function approximator as well, but the input is the action chosen by the actor and the output is the Q value of this action - i.e.: essentially the maximum future reward. Finally, the training of the nets is performed separately and the *weights are updated at each time step, at the end of the i-th trajectory collecting phase (rollout)* - a process called temporal difference learning (TD Learning).
In the following the main actor critic methods that have been used in this work are briefly introduced.

2.6.2 *Trust Region Methods (TRM)*

Notwithstanding their performances, actor-critic methods face two major drawbacks:

(i) *Unstable Update*: The process of learning the policy is fundamental in order to achieve good results. In a context of gradient assent if the learning rate is too small the overall learning process

would be slowed down. On the other hand, a overly large learning rate compromises learning, missing important trajectories.

Figure 11: Policy Stable Update, from J. Hui post in [10]

(ii) *Data inefficiency*: The data collected evaluating a policy is lost right after gradient update. That means that every time a new policy is created, this process must be repeated from the beginning. Consequently, considering the amount of information needed for a NN to be optimised, a major slowdown of the process is obtained.

In order to face the unstable update, a constraint on how much the policy can be updated for each step is imposed. Since the policy is essentially a probability distribution, to compute the distance between the current one and the updated one the *Kullback - Leibler Divergence* is used. Considering two adjacent policies $\pi_1(a|s)$ and $\pi_2(a|s)$, the KL-Divergence is computed as follows:

$$D_{KL}(\pi_1 \| \pi_2)[s] = \sum_{a \in \mathcal{A}} \pi_1(a|s) \log\left(\frac{\pi_1(a|s)}{\pi_2(a|s)}\right) \tag{14}$$

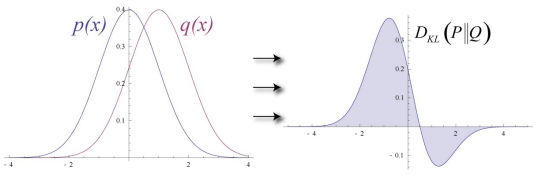

Figure 12: KL-Divergence example on two PDFs

To tackle data inefficiency, a *replay buffer* containing data collected by another policy distribution is created. Here it comes to play a sta-

tistical tool called *importance sampling*. The new surrogate objective function can be derived:

$$J(\theta) = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta,old}}[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta,old}(s_t|a_t)}A(s_t,a_t)] \tag{15}$$

It can be seen how this new cost function features the ratio of the old ($\pi_{\theta,old}$) and the new (π_θ) policy.
Finally, it has been mathematically proved from the Appendix in [27] that *trust region optimisation guarantees the monotonic policy improvement*.

2.6.2.1 *Proximal Policy Optimisation (PPO)*

Belonging to the Trust Region Methods (TRM), Proximal Policy Optimisation outperforms other methods for continuous control problem. That is especially true for the version with a clipped objective. The idea is firstly proposed in [28]. Essentially the ratio between the old policy and the new one is clipped in a certain proximal range.

$$max_\theta \hat{\mathbb{E}}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta,old}(a_t|s_t)}\hat{A}_t] \quad r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta,old}(a_t|s_t)}$$

If the probability ratio ($r_t(\theta)$) exceeds a region defined by $[1-\epsilon, 1+\epsilon]$, the objective is clipped. The new *clipped* objective function is:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t] \tag{16}$$

During the implementation phase, a more complex objective function is used, adding a squared error loss of the critic and an entropy bonus to ensure sufficient exploration.

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP} - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{17}$$

where:

- L_t^{VF} is the squared-error loss for the critic network: $L_t^{VF} = (v_\theta(s_t) - V_t^{targ})^2$
- c_1, c_2 are two constants. Changing their value conditions the actor and critic losses. c_2 is also called *entropy parameter*. Increasing it will encourage a more wider exploration.

Algorithm 1: PPO, Actor-Critic Style [28]

Initialise the *Actor* network $\pi^{\theta}(s)$ and *Critic* network $V_{\pi}^{\omega}(s)$ with weights θ and ω;
for *iteration=1,2,...* **do**
  for *actor=1,2,..., N* **do**
    Run policy $\pi_{\theta,old}$ in $\mathcal{E}$ for T timesteps ;
    Compute advantage estimates $\hat{A_1}, ..., \hat{A_T}$;
  end
  Optimise surrogate L wrt θ;
  $\theta_{old} \leftarrow \theta$;
end

It has to be noted how this method descends from *Trust Region Policy Optimisation (TRPO)* from J. Shulman et al [27]; it can be even interpreted as a first-order approximation of TRPO. Due to its better empirical performances, the PPO will be preferred to TRPO in this work.

PPO is one of the most widely used RL algorithm by the community. For instance, OpenAI - a leading company in the RL research field - refers to having made PPO their default RL algorithm inside their framework for its ease of use and performance. Also different published works that use RL in a variety of applications often turn to PPO, for instance see its application in Fluid Mechanics by J. Rabault et al. [24]. For this reason, it will also be the algorithm of choice in this work.

2.7 HYPER PARAMETERS OPTIMISATION: BOOSTING RL OR A STANDALONE ALTERNATIVE?

Definition 2.7.1. In any machine learning framework, an *hyperparameter* is a parameter initialised before training a model.

These are not to be confused with *model parameters* that are a property set during the learning process. An example of model parameters are the weights and biases of a NNs, while an example of hyperparameter is the learning rate.

Any Machine Learning framework has its own hyperparameters and how to properly set them in order to maximise the performance of a given algorithm is a crucial task carried out by what is called *Auto Machine Learning (ML)* - this is especially true when dealing with neural networks. Hence, hyper parameters optimisation (HPO) is useful to:

- Reduce the human effort in setting the model up to the simulation;

- Improving the performance of the algorithm *for the problem at hand*, tailoring the right set of hyperparameters can lead to state-of-the-art performances.
- Improve reproducibility and comparisons between different methods.

HPO is not anew, it has a long history and now with the increasing attention of the scientific community into Machine Learning it has returned to be focus of attention by many.

However, there are several challenges in applying HPO:

(i) When dealing with deep learning, function evaluation may be very expensive to evaluate;

(ii) The hyper parameters space is often high dimensional and complex (containining a mix of continuous, conditional and categorical hyperparameters). Moreover, not all the hyperparameters influence the objective function in the same way, and deciding which is worthy to be tuned and which is not is not trivial;

(iii) Generally speaking, properties used in classical optimisation - such as convexity and smoothness - are not applicable to this cases;

(iv) The gradient of the loss function with respect to the hyperparameters is not usually available.

The structure of the chapter is the following: at first, the HPO problem will be mathematically stated. After that, a review of the main HPO method that will be used in this work will be presented. Lastly, a brief overview of the most important hyper parameters in the Reinforcement Learning methods presented in 2.4 is given.

For more insights about HPO and its influences on machine learning problems, the reader can refer to the Auto ML book written by the researchers of the universities of Freiburg and Hannover, freely available [11].

2.7.1 *Problem Statement*

Let $\mathcal{A}$ be a machine learning algorithm with N hyperparameters. The domain of the $n-th$ hyperparameter is denoted as Λ_n and the overall *hyper parameter space* as:

$$\boldsymbol{\Lambda} = \Lambda_1 \times \Lambda_2 \times ... \times \Lambda_N$$

A vector of hyper parameters is defined as $\lambda \in \boldsymbol{\Lambda}$ and $\mathcal{A}$ using hyperparameters λ is denoted as $\mathcal{A}_\lambda$.

The loss of a model defined by algorithm $\mathcal{A}$ with hyperparameters λ on a training dataset $\mathcal{D}_{train}$ is defined as:

$$\mathbf{V}(\mathcal{L}, \mathcal{A}_\lambda, \mathcal{D}_{train})$$

where $\mathcal{L}$ is the loss function of the problem at stake.
Finally, the Hyper Parameters Optimisation problem can be stated as follows:

$$\lambda^* = \mathrm{argmin}_{\lambda \in \Lambda} \mathbb{E}_{\mathcal{D}_{train} \sim \mathcal{D}} \mathbf{V}(\mathcal{L}, \mathcal{A}_\lambda, \mathcal{D}_{train}) \tag{18}$$

It has to be noted that if the loss function is evaluated in a dataset different from the training one - operation called *cross validations* - the definition changes:

$$\lambda^* = \mathrm{argmin}_{\lambda \in \Lambda} \mathbb{E}_{\mathcal{D}_{train}, \mathcal{D}_{valid} \sim \mathcal{D}} \mathbf{V}(\mathcal{L}, \mathcal{A}_\lambda, \mathcal{D}_{train}, \mathcal{D}_{valid})$$

Now how to sample the hyper parameter vector $\lambda \in \Lambda$ is discussed.

2.7.2 *Bayesian Optimisation (BO)*

Bayesian Optimisation is by far the most used Hyper Parameters Optimisation tool, outperforming in terms of robustness - i.e.: results have a low sensitivity to the method parameters - other methods tried for facing the HPO challenge applied to this work.

2.7.2.1 *Bayesian Optimisation in a Nutshell*

This section gives a brief insight on the internal working of the algorithm. For a more complete discussion and overview, the reader is referred to the tutorial of Shahriari et al in [29].

Bayesian Optimisation relies on a *probabilistic surrogate model* and on a *acquisition function*. The first is a mapping of the current hyperparameters vector λ to a probability of score on the objective function r, it is denoted as:

$$\mathcal{P}(r|\lambda)$$

$\mathcal{P}$ is also called *surrogate* of the original objective function and it is easier to optimise with respect to the latter. The algorithm essentially suggests successive hyperparameters vectors that have a good performance on the *surrogate* by mean of the *acquisition function*. Although there are several acquisition functions available to be tested, most often the *expected improvement* (EI):

$$\mathbb{E}[\mathbb{I}(\boldsymbol{\lambda})] = \mathbb{E}[\max(f_{min} - y, 0)] \tag{19}$$

Is often chosen since it follows a normal distribution:

$$\mathbb{E}[\mathbb{I}(\boldsymbol{\lambda})] = (f_{min} - \mu(\boldsymbol{\lambda}) \phi\Big(\frac{f_{min} - \mu(\boldsymbol{\lambda})}{\sigma}\Big) + \sigma \mathcal{N}(\frac{f_{min} - \mu(\boldsymbol{\lambda})}{\sigma}\Big) \tag{20}$$

Where:

- $\phi(x)$: is the probability density function;
- $\mathcal{N}$: is the standard normal distribution;
- f_{min}: is the best score observed so far;
- μ: is the mean of the distribution;
- σ: is the square of the ditribution variance;
- $\boldsymbol{\lambda}$: is the current set of hyperparameters;

The acquisition function, proposing new points to be sampled, implicitly deals with the *exploration-exploitation dilemma*, a common challenge in the ML community already presented in Sect. 2.4. Here, the *exploration* try to find new points in regions where the uncertainty is high, in order to try to explore all the available space. On the other hand, *exploitation* suggests promising spots basing on the score obtained on the *surrogate*. Dealing with the proper balance of the two is a delicate matter and failing in this task could lead to being stuck in local minima.

Finally, the *surrogate objective function* has to be defined. Usually, *Gaussian Processes* $\mathcal{G}$ (GP) are employed. A Gaussian process is a stochastic process defined as:

$$\mathcal{G}(m(\boldsymbol{\lambda}), k(\boldsymbol{\lambda}, \boldsymbol{\lambda}^{'})) \tag{21}$$

Where:

- $m(\boldsymbol{\lambda})$: is the *mean*;
- $k(\boldsymbol{\lambda}, \boldsymbol{\lambda}')$: is the co-variance function.

Gaussian processes are often preferred to other alternatives because of their smoothness and expressiveness. Moreover, the choice of representing the objective function as a probability density function has the benefit that it can be successively updated following probabilistic *Bayesian* processes. However, GP properties are subject to the choice of the covariance function. The most common choice in the literature is the *Mâtern 5/2 kernel*. Notwithstanding their reliability, GP have also some drawbacks. The most important one is that they scale cubically with the number of data points. Hence, their applicability is strictly linked with the possibility of doing several function evaluations. This problem can be avoided using *sparse* GP, but the discussion of these methods is out of scope of this brief overview.
Finally, the algorithm may be generically written as follows.

By now, it should be clear how this optimisation technique belongs to *Bayesian* statistics. The core of this algorithm is updating a *prior* - i.e.: previous - belief accordingly to new information sampled and finally generating a *posterior* - i.e.: successive.

Algorithm 2: Bayesian Optimisation Algorithm

Initialise the Gaussian Process *surrogate function* prior distribution;
for *each iteration* **do**
  Select new x_{n+1} optimising the acquisition function α;
  $x_{n+1} = \operatorname{argmax}_x \quad \alpha(x; \mathcal{D}_n)$;
  Query objective function to obtain y_{n+1};
  Update the Gaussian process prior distribution with new data to produce a posterior;
  Interpret the current Gaussian Process distribution to fin the global minima;
end

In order to summarise the previous review on Bayesian Optimisation, Fig. 13 shows an example of Bayesian Optimisation at work is shown for a 1D function.

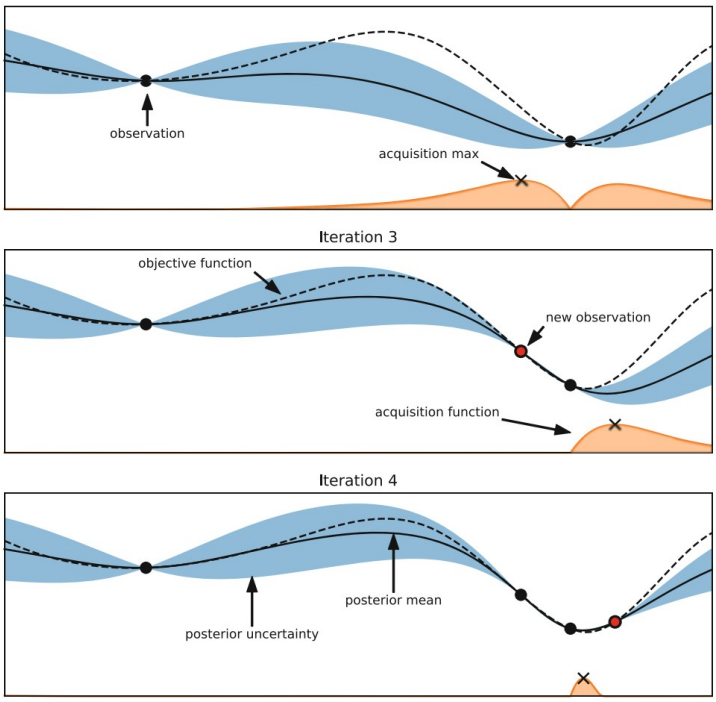

Figure 13: Example of Bayesian Optimisation on a 1D function, from [11]

The goal is to minimise the dashed black line by means of the Gaussian Process surrogate. The acquisition function has an high value around points where the predicted function value is relatively low and the uncertainty (light blue area in the picture) is high. On the other hand, it has a low value nearby observation points i.e.: points already sampled. It can be appreciated how, going on with the iterations, the objective function is gradually reduced and how there is almost no uncertainty in proximity of the minima.

2.7.3 *Main Reinforcement Learning Hyper Parameters*

Taking over the description of the main algorithm that is going to be used - PPO - in Sect. 2.6, the description can be now enriched with a concise description of its main hyper parameters.

Hyper Parameters (from [8])

Name	Description
PPO	
Discount factor (γ)	defines the discounting of future rewards
Number of steps	steps to run for each trajectory (*rollout*)
Learning rate (α)	determines the gradient assent step size
VF coeff.	Value function coefficient in the loss calculation
Cliprange	clipping parameter of the objective function
nminibatches	defines the batches subdivision for learning
Ent. Coeff.	Entropy coefficient for the loss calculation
Lam	Factor for trade-off of bias vs variance for GAE [a]

Table 1: Hyper Parameters list

[a] Generalized Advantage Estimator

It can be understood how an optimisation with such a number of parameters is computationally heavy. Moreover, the objective in this optimisation is the *collected reward* by the agent during a simulation that also requires its share of computational time.

Despite these challenging aspects, HPO has the chance of enhance the performance of any Machine Learning model - from Deep Learning to Deep Reinforcement Learning - properly adjusting these parameters to the problem at stake. Again, it has to be kept in mind that does not exist a set of hyperparameters "globally optimal" but these have to be tailored on the problem at stake. On the other hand, however, this technique may literally boost the performances of an algorithm.

This is a so important challenge in the Machine Learning community that renowned software companies such as Microsoft ([33]) or Facebook ([6]) have released open-source their optimisation toolkits.

2.7.4 *Changing the Perspective: BO for optimal control*

Changing a bit the perspective followed so far, and looking at the problem from a more high-level standpoint, the HPO problem is nothing more than a constrained optimisation. In fact, we have a constraint - usually more than one - represented by the boundaries of the parameters space to be searched and the purpose is to maximise some objective function. It can be noticed how this definition well suits a general control problem, such as the one assessed in this work. In fact, recalling eq. (18):

$$\lambda^* = \mathrm{argmin}_{\lambda \in \Lambda} \mathbb{E}_{\mathcal{D}_{\mathtt{train}} \sim \mathcal{D}} \boldsymbol{V}(\mathcal{L}, \mathcal{A}_\lambda, \mathcal{D}_{\mathtt{train}})$$

From the control point of view, now these elements change their meanings but not their roles: λ^* becomes an optimal set of control coefficients that, combined with the observed state, give birth to an action that minimise a given cost-functional $\mathbb{E}_{\mathcal{D}_{\mathtt{train}} \sim \mathcal{D}} \boldsymbol{V}(...)$, objective of the control problem at stake. For instance, following the DRL definition these can be the *reward* of the controller interacting with the system.

Therefore, it would be interesting if instead of using these methods in order to optimise the DRL agent parameters that performs the control they might be used to directly perform the control by themselves. This idea will be applied in the following in order to assess its applicability - in the limit of the test case hereafter examined.

2.8 LINEAR REGRESSION OF RL STRATEGIES

Having applied the steps described so far, a *control-law* learnt by the agent should be now available. Driven by curiosity, the next logical step is to make a comparison of the control laws derived via RL and other approaches.

Previously, in Sect. 2.4 it has been shown how the Deep Reinforcement Learning uses a neural network to map states in to actions, following the actual policy available to the agent. Further, when neural networks were briefly introduced in Sect. 2.3, it has been explained how these are non linear by definition and how these are *universal approximators*: that is, these are able to learn any function if a proper number of layers is given.

Neglecting such complexities, an attempt in order to study these laws is conducted assuming naively a linear control law - i.e.: a linear relationship between the input (or observations) and outputs (or actions). That is, assuming that:

$$\underbrace{\mathbf{O}}_{t\times \#obs} \cdot \overbrace{\mathbf{C}}^{\#obs\times a} = \underbrace{\mathbf{A}}_{t\times a} \tag{22}$$

where O is the input array containing the states observations, A is the actions array and C is the array containing the *linear* control coefficients. The dimensions of these matrices are defined by the simulation: t represents the time span in which the controller is active, $\#obs$ is the size of the input - i.e.: the states that are passed to the agent - and a stands for the actions made by the controller.

As a next step, given a set of observations O and actions A taken by the RL agent, it is of interest to find the best set of coefficients C. This is a least square problem. However, the system might be harshly not squared, and matrix inversion is not defined for matrices that are not square. Further, depending of the dimensions of $\mathbf{O}$ there could be multiple solutions, if it is wider than tall, or no solutions at all, if it is taller than wide.

The *Moore-Penrose pseudoinverse* allows these cases to be approximately solved. The pseudoinverse of a generic matrix $\mathbf{A}$ can be defined as

$$\mathbf{A}^{+} = \mathbf{V}\mathbf{D}^{+}\mathbf{U}^{T} \tag{23}$$

where $\mathbf{U}$, $\mathbf{D}$ and $\mathbf{V}$ are the matrices corresponding to the singular value decomposition of $\mathbf{A}$, the pseudoinverse $\mathbf{D}^{+}$ of a diagonal matrix $\mathbf{D}$ obtained by taking the reciprocal of its nonzero elements. The definition in eq. (23) is also used in the `NumPy` algorithm used as explained in [22].

For a more detailed discussion of the Moore-Penrose pseudoinverse see J.C.A Barata et al. in [3].

Depending on the dimensions of the starting matrix, the pseudoinverse assumes different meanings. For instance, considering a general system:

$$Ax = y$$

i. **A has more columns than rows:** in this case the linear system would have many solutions. Hence, the minimal norm solution exploiting the pseudoinverse returns the single solution

$$\boldsymbol{x} = \boldsymbol{A}^{+}\boldsymbol{y}$$

ii. **A has more rows than column:** in such a case it is possible that the system may have no solutions at all. The solution yielded by solving the system with the pseudoinverse here is the solution for which $\boldsymbol{Ax}$ is the closest to $\boldsymbol{y}$ in a terms of *Euclidean norm*

$$\|\boldsymbol{Ax} - \boldsymbol{y}\|_2$$

Our situation is surely part of the latter situation, being the $t \gg \#obs$. Hence applying the pseudoinverse in our case would lead to a *array of control coefficients* defined as:

$$\mathbf{C}_{pinv} = (\mathbf{O}^{T}\mathbf{O})^{-1}\mathbf{O}^{T}\mathbf{A} \tag{24}$$

This solution yields to a *least-square approximation* of the control law, that is presumably non linear, descending by a non linear mapping operator such as a neural network. Thereafter, the *least-square actions* may be computed as:

$$\mathbf{O} \cdot \mathbf{C}_{pinv} = \mathbf{A}_{approx} \tag{25}$$

Now that both the control action applied by a deep reinforcement learning controller and a linear one are available, an comparison between the two can be conducted. This procedure might show at glance the differences between these radically different approaches.

2.9 OTHER OPTIMISATIONS TECHNIQUES: A BRIEF REVIEW

In addition to Bayesian Optimisation, other optimisations techniques are used in this work. This section provides a general overview of them.

2.9.1 *Nelder-Mead method (or Simplex Method)*

The Nelder-Mead [18] method is a non linear optimisation technique based on a domain of n dimensions. One of the key features of this

method is that it is *gradient free*. It belongs to the *direct search* category, which is based on function comparison. The core of this technique is a *simplex*, a polytope of $n+1$ vertices in n dimensions. The objective function is computed in each vertex point, in order to search for a new candidate point. Then a *centroid point* is computed discarding the worst test point: if it is better of the best test point then the polytope is stretched along this direction. Otherwise, a *valley* is reached and so the simplex is shrunken, searching for a new best point, and so on.

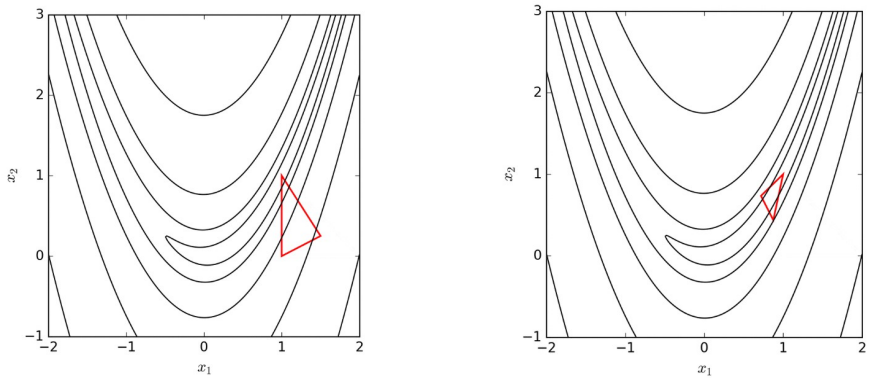

Figure 14: Two successive steps of the Nelder-Mead algorithm applied to the Rosenbrock function

Standard values of the empirical coefficients appearing in algorithm 3 are:
$\alpha = 1, \quad \gamma = 1, \quad \rho = 1/2, \quad \sigma = 1/2.$
Usually, if the standard deviation of the current simplex values are in a tolerance the cycle is stopped, even if other exit conditions might be imposed.

2.9.2 *Lipschtiz functions method*

This *parameter-free* optimisation method, proposed by Malherbe et al. [15] relies on the computation of the *piece wise upper bound of* $f(x)$ in order to find a new set of x to evaluate.

Fig. 15 shows an example of this application, with the upper limiter in green.

$$U(x) = \min_{i=1,\dots,t} (f(x_i) + k\|x - x_i\|_2) \tag{26}$$

Algorithm 3: Nelder-Mead Minimization Algorithm

Trying to minimize the function $f(x)$;
Current test points $x_1, ..., x_{n+1}$;
for *trials* = *1,..., N* **do**
  1. *Order* values at the vertices: $f(x_1) \leqslant f(x_2) \leqslant ... \leqslant f(x_{n+1})$;
  2. *Centroid* x_0 updated computation, including all points except x_{n+1};
  3. *Reflection*: compute the new reflected point $x_r = x_0 + \alpha(x_0 - x_{n+1})$ $(\alpha > 0)$;
  if $f(x_r) > f(x_i)$ **then**
    Update the new simplex vertices, substituting the worst point x_{n+1}, then go to step 1.;
  end
  4. *Expansion*: If the reflected point is the best point computed, $f(x_r) < f(x_1)$ then compute the extended point $x_e = x_0 + \gamma(x_r - x_0)$ with $\gamma > 1$;
  if $f(x_e) < f(x_r)$ **then**
    Update the new simplex vertices, substituting the worst point x_{n+1} wit x_e, then go to step 1.;
  end
  else
    Update the new simplex vertices, substituting the worst point x_{n+1} with x_r, then go to step 1.;
  end
  5. *Contraction*: At this step, $f(x_r) \geqslant f(x_n)$;
  Compute the contracted point $x_c = x_0 + \rho(x_{n+1} - x_0)$ where $0 < \rho \leqslant 0.5$;
  if $f(x_c) < f(x_{n+1})$ **then**
    Update the new simplex vertices, substituting the worst point x_{n+1} with x_c, then go to step 1.
  end
  6. *Shrink*: replace all points except the best x_1 with $x_i = x_1 + \sigma(x_i - x_1)$ and go to step 1.
end

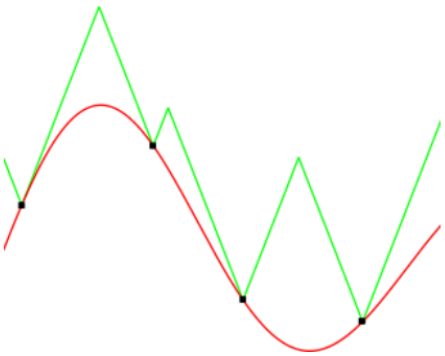

Figure 15: Example of Lipschtiz function application

Algorithm 4: LIPO basic steps

```
Initalisation with random x;
for trial = 1, N do
    Compute the upper bound of x
    if U(x) > U(x)_best then
        x is the new point to evaluate;
        end
    end
```

This simple algorithm of Alg. 4 is then further expanded comprehending several dimensions space and evaluating the importance of each parameter - i.e.: computing a k per each.

Without dive too much inside this algorithm, it is anticipated that it will be especially used in the first steps of the optimisation carried out in Sect. 3.6. In fact, this algorithm has the ability to identify a region in which the local optima are to be found.

3 TEST CASES AND IMPLEMENTATIONS

This section presents the test cases analyzed in this work. Moreover, the details of the implementations are explained, for the sake of reproducibility.

3.1 REINFORCEMENT LEARNING WITH STABLE BASELINES

As a framework for the experiments with deep reinforcement learning, *Stable Baselines* [8] is used. It descends from OpenAI Baselines [8] and implements cutting-edge algorithms. Having a common *baseline* in conducting Reinforcement Learning tests allows a more efficient replication of results and ideas circulation.

Moreover, these frameworks are useful to unburden the weight of low-level implementation issues, making easier to conduct a RL-based research.

In order to make the RL-agent to interact with a given problem, one must formulate such a problem in an *environment* (see Sect. 2.4 for theoretical reference). A custom environment must have the following structure:

```
import gym
from gym import spaces

class CustomEnv(gym.Env):
  """Custom Environment that follows gym interface"""
  metadata = {'render.modes': ['human']}

  def __init__(self, arg1, arg2, ...):
    super(CustomEnv, self).__init__()
    # Define action and observation space
    # They must be gym.spaces objects
    # Example when using discrete actions:
    self.action_space = spaces.Discrete(N_DISCRETE_ACTIONS)
    # Example for using image as input:
    self.observation_space = spaces.Box(low=0, high=255,
                                        shape=(HEIGHT, WIDTH, N_CHANNELS)
    , dtype=np.uint8)

  def step(self, action):
    ...
    return observation, reward, done, info
  def reset(self):
    ...
    return observation  # reward, done, info can't be included
  def render(self, mode='human'):
    ...
  def close (self):
```

```
    ...
```

Python Code 1: Designing a Custom Environment with Stable Baselines

Therefore, a custom environment is organised as a *class* (see Python docs for their peculiarities [20]) and it is composed by different blocks - or, following the python proper terminology, *functions* ([21]):

- **`def __init__(self, arg1, arg2, ...)`**: this function initialises the main parameters that will be used throughout the simulation;
- **`def step(self, action)`**: this is function is the core of the simulation. The agent is interacting with the environment applying an `action`. From this interaction, it receives the new state of the system (`observations`), the `reward` linked with such an action and if the game is finished (`done`). Moreover, `info` can give back some additional information;
- **`def reset(self)`**: this function resets the simulation to its starting point;
- **`def render(self, mode = 'human')`**: if desired, the simulation results may be shown through this function.
- **`def close(self):`** correctly closes the simulation.

After implementing the environment, the simulation can be launched:

```
# Instantiate the env
env = CustomEnv(arg1, ...)
# Define and Train the agent
model = ALGO('Policy', env).learn(total_timesteps=N)
```

Python Code 2: Starting a simulation with Stable Baselines

where:

- **`ALGO`:** is the algorithm that is going to be used. Stable Baselines offers a wide collection of methods available, including A2C, PPO and SAC (explained in 2.6) and many others;
- **`'Policy'`:** this argument defines how the policy is going to be mapped, essentially it is the architecture of the net. The main two architectures that are going to be tested are:
  i. MLP: Multi Layer Perceptron;
  ii. MLP Lstm: MLP with LSTM network for feature extraction.
- **`learn(total_timesteps=N)`**: is a method defining the length of the training phase, setting a constraint on the total agent-environment interactions to be tested.

3.1.1 *Custom Callback: dynamically stopping the simulation*

As explained above, in the simplest implementation the model receives as an input the number of interactions it is going to have with environment.

Another approach consists in stopping the training as soon as some condition is reached, for instance the rewards obtained by the agent are approximately constant with time. In fact, that would mean that the agent has learnt the policy and there is little modification going on in the policy. In this case, the agent is going to *exploit* what has learnt so far.

This is done using a *custom callback* that interacts with the data extracted by the simulation:

```
def _on_step(self) -> bool:

    if self.num_timesteps > K*self.eval_freq: # to be sure that var != 0
        self.activate = True

    episode_rewards, episode_lengths = evaluate_policy(self.model, ...)

    mean_reward, std_reward = np.mean(episode_rewards), np.std(
    episode_rewards)
    self.log_reward.append(mean_reward)
    var = np.var(np.asarray(self.log_reward))

        if self.activate and var < self.treshold:
            '''
            If there is not a variation in the collected reward of the
    treshold% minimum then the simulation is stopped
            '''
            ...
            return False
```

Python Code 3: Custom Callback

This piece of code (the full script may be found in the GitHub repository) essentially is doing what explained above: during each step (`def _on_step()`), the mean reward are stocked inside a deque list (Doubly Ended Queue) called `self.log_reward` of a given length that automatically erases the oldest data in memory if its size is exceeded. Finally, if the variance of the data stocked is lower than a user imposed constraint (`self.treshold`) the simulation is stopped.

In order to avoid the variance to be zero and neglecting the first noisy interactions with the environment, this whole process starts after `K*self.eval_freq` steps, always defined by the user.

Therefore, using this tool, there is the reasonable certainty that the simulation will go on until it is needed to the agent to learn, giving back also a metric of the learning difficulty and/or the algorithm efficiency.

3.2 HYPER PARAMETERS OPTIMISATION WITH OPTUNA

For what concerns the Hyper Parameters Optimisation (HPO), discussed in 2.7, the Optuna framework, developed by T. Akiba et al. [2], will be the implementation of choice. This tool is widely recognised as one of the most effective implementation of the *Bayesian Optimisation* algorithm and it is largely used by the whole machine learning community.

The paper released by the developers contains the main features included.

This black box optimisation technique essentially requires only a given objective function to pursue and the boundaries of the hyper parameter space to search into in order to start an optimisation - or *study*. For instance, the following straightforward example may be considered:

```
import optuna

def objective(trial):
    x = trial.suggest_uniform('x', -10, 10)
    return (x - 2) ** 2

study = optuna.create_study()
study.optimize(objective, n_trials=100)

study.best_params
```

Python Code 4: Optuna example

Among the various features included in such a package, one that is particularly appreciated is the ability to define the *search space typology*. For instance, the user can define if the next value to be suggested by the *acquisition function* has to be chosen following a continue, discrete or even logarithmic distribution. This possibility can lead to major computational advantages when the topology of the space to be searched for a given parameter is known. This is the case for some of the parameters to be optimised in the following test cases.

Finally, it can be noticed that the objective to be minimised can embody the whole system dynamics - i.e.: the total control problem can be assigned to such an optimiser in order to be attacked.

3.3 OPTIMIZER IMPLEMENTATIONS

Both for Lipschtiz functions (Sect. 15) and for Nelder-Mead (Sect. 2.9) two stable open source implementations are used.

3.3.1 *Nelder-Mead*

Nelder-Mead is one of the many minimization techniques that `SciPy` offers in its `minimize` method, for more information please refer to [17]. The implementation of such a method for optimize the Rosen function, as an example, is the following:

```
from scipy.optimize import minimize, rosen, rosen_der

>>>x0 = [1.3, 0.7, 0.8, 1.9, 1.2]

res = minimize(rosen, x0, method='Nelder-Mead', tol=1e-6)

res.x
array([ 1.,  1.,  1.,  1.,  1.])
```

Python Code 5: SciPy Nelder Mead

Where `x0` is he initial guess for each value, and `tol` specifyies when the simulation has to be stopped.

3.3.2 *Lipschiz functions with Dlib: LIPO*

The implementation of choice of this algorithm is the one proposed in the C-based library Dlib in [12]. For a given function to be optimized, an example of the implementation could be:

```
import dlib

def holder_table(x0,x1):
    return -abs(sin(x0)*cos(x1)*exp(abs(1-sqrt(x0*x0+x1*x1)/pi)))

x,y = dlib.find_min_global(holder_table,
                           [-10,-10],
                           # Lower bound constraints on x0 and x1
                           [10,10],
                           # Upper bound constraints on x0 and x1
                           80)
                           # Number of trials
```

Python Code 6: Dlib LIPO

3.4 TEST CASE N.1 - LINEAR ADVECTION

3.4.1 *Environment description*

The *Advection equation* is a first order *linear* PDE (partial derivative equation). This equation is hyperbolic and can be explicitly solved using the *method of characteristics*.

This equation reads:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0, \tag{27}$$

where c is a non zero wave propagation speed. This equation then describes the motion of a scalar u as it is advected by a known velocity, c.

In this test case, a finite - difference discretisation was used. Following this approach, the computational domain is divided in a set of points where the solution is stored. As a convention, the discretised generic function will be expressed as $a_i = a(x_i)$ where $x(i)$ is the i-th point of the discretised domain. Recalling 27:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0$$

Then, a first order discretization forward in time backward in space would be:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = -u \cdot \frac{u_i^n - u_{i-1}^n}{\Delta x} \tag{28}$$

However, first order methods are not very satisfactory in a retained error point of view. For that reason, a second order accurate *Lax-Wendoff scheme* has been implemented. Before writing the formulas it has to be remembered that higher order methods would imply the computation of the Jacobian matrix. In order to avoid this step, that is usually computationally expensive, that would slow down the simulations, the *Richtmyer method* is applied. This method evaluates the Lax - Wendroff method in two steps:

$$\begin{cases} u_{i+1/2}^{n+1/2} = \frac{1}{2}(u_{i+1}^n + u_i^n) - \frac{\Delta t}{2\Delta x}(f(u_{i+1}^n) - f(u_i^n)) \\ \\ u_{i-1/2}^{n+1/2} = \frac{1}{2}(u_i^n + u_{i-1}^n) - \frac{\Delta t}{2\Delta x}(f(u_i^n) - f(u_{i-1}^n)) \end{cases} \tag{29}$$

And finally computing

$$u_i^{n+1} = u_i^n - \frac{\Delta t}{\Delta x}[f(u_{i+1/2}^{n+1/2}) - f(u_{i-1/2}^{n+1/2})] \tag{30}$$

Finally, in order to properly pose the discretised problem, the *boundary conditions* have to be imposed. The *Neumann boundary conditions*

will be applied, also known as boundary condition of the second kind. This typology of boundary condition prescribes the gradient of the variable normal to the boundary. Generally,

$$\begin{cases} \frac{d\phi}{dx}|_{x=x_0} = \alpha \\ \frac{d\phi}{dx}|_{x=x_L} = \beta \end{cases} \tag{31}$$

Where α and β are indeed the prescribed values. In this test case these values will be imposed equal to zero, hence leading to *homogeneous* boundary conditions:

$$\begin{cases} \alpha = 0 \\ \beta = 0 \end{cases}$$

After these preliminary considerations, the problem may be introduced as follows:

$$\frac{\partial a}{\partial t} + a\frac{\partial u}{\partial x} = \underbrace{f(x,t)}_{\text{disturbance}} + \underbrace{g(x,t)}_{\text{control}} \tag{32}$$

Where f no longer represents the flux as in Eq. 29 but it is the disturbance:

$$\underbrace{f(x,t)}_{\text{disturbance}} = A\sin(\omega t)\cdot\mathcal{N}(x-5, 0.2) \tag{33}$$

and

$$\underbrace{g(x,t)}_{\text{control}} = \text{action}(t)\cdot\mathcal{N}(x-18, 0.2) \tag{34}$$

Having placed the disturbance at $x = 5$ and the control action at $x = 18$.
Finally, the observation points and the reward are defined as follows:

$$\underbrace{\mathcal{O}(u)}_{\text{observations}} = \bar{u}(t-1), \bar{u}(t-2), \bar{u}(t-3) \tag{35}$$

Where $\bar{u}$ is the state observed at time t.

This control problem has an exact solution given by the *time-shifted sine wave*.

$$\underbrace{g(x,t)}_{\text{control exact}} = A\sin\omega t - \omega\frac{\Delta x}{c} + \pi\cdot\mathcal{N}(x-(5+\Delta x), 0.2) \tag{36}$$

where

- ω = pulsation

- Δx = distance between sources
- c = propagation speed

In this test case, firstly posed in [19], a simple 1D advection equation with constants coefficients is disturbed by a Gaussian perturbation in space and a sinus pulsating harmonically in time. The control action is made by the RL agent through a Gaussian in space as well. The action that the agent is going to make is defined by a policy that bases its output on *observations* of the environment and the *reward* obtained. The previous have been designed as follows:

- *Observations*: the states given back to the agent are six spatial points at three different time steps. In particular, the observations at t, $t-1$ and $t-2$.
- *Reward*: In this first approach it is designed as an Euclidean norm of the displacement in a space portion after the Gaussian control.

$$\mathcal{R} = \|u\|_2$$

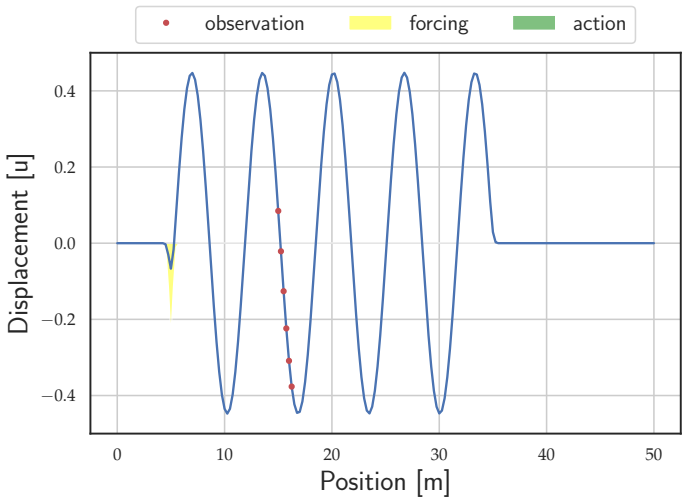

Figure 16: Advection Equation environment rendering without control

The final goal for the RL agent is to cancel out the harmonic perturbation generated by the disturbance, relying only on the information given by the observation points and trying to minimise the reward shaped as already described.

In this first linear test case, the efficiency of the algorithm will be assessed comparing the control law learnt by the agent with the theoretical one, presented in eq. (36).

3.4.2 *Implementations*

Transposing what said so far in Python code, the advection equation may be written as:

```
#--------------
# Advance simulation one time step
#--------------
u_half_1 = 0.5*(self.u1[1:-1] + self.u1[:-2]) +
          - (self.C/2)*(self.u1[1:-1] - self.u1[:-2])
u_half_2 = 0.5*(self.u1[1:-1] + self.u1[2:]) +
          - (self.C/2)*(self.u1[2:] - self.u1[1:-1])

self.u[1:-1] = self.u1[1:-1] +
              - (self.c*self.dt/self.dx)*(u_half_2 - u_half_1)

# Boundary Conditions
self.u[0] = self.u1[1]; self.u[-1] = self.u1[-2]

# Switch variables before next step
self.u3[:], self.u2[:], self.u1[:] = self.u2, self.u1, self.u
```

Python Code 7: Advection time stepping

Two terms were added to the RHS of the time-marching step, to include the perturbation and the control action. The result reads:

```
    #--------------
    # Advance simulation one time step
    #--------------
    u_half_1 = 0.5*(self.u1[1:-1] + self.u1[:-2]) +
              - (self.C/2)*(self.u1[1:-1] - self.u1[:-2])
    u_half_2 = 0.5*(self.u1[1:-1] + self.u1[2:]) +
              - (self.C/2)*(self.u1[2:] - self.u1[1:-1])
    self.u[1:-1] = self.u1[1:-1] +
              - (self.c*self.dt/self.dx)*(u_half_2 - u_half_1) +
              + self.dt*(fun[1:-1] + action_vec[1:-1])

    # Boundary Conditions
    self.u[0] = self.u1[1]; self.u[-1] = self.u1[-2]

    # Switch variables before next step
    self.u3[:], self.u2[:], self.u1[:] = self.u2, self.u1, self.u
```

Python Code 8: Advection time stepping with external forces

It has to be noticed that the term denoted with C is the *Courant number*:

$$C = c\frac{\Delta t}{\Delta x} \tag{37}$$

This number is linked with the *stability* of the numerical scheme. In fact, recalling briefly the *Courant - Friedrichs - Lewy* (CFL) condi-

tion, that is necessary condition for convergence of explicit numerical methods for solving hyperbolic PDEs. The condition imposes

$$C = c\frac{\Delta t}{\Delta x} \leqslant C_{max} \tag{38}$$

Where C_{max} changes if we are using an explicit or implicit method. In this test case, where we are using an explicit method,

$$C_{max} = 1$$

From its definition in eq. (37) it can be easily reversed in order to obtain a time step that will satisfy this very stability condition:

$$\Delta t = C\frac{\Delta x}{u} \tag{39}$$

It can be shown that the CFL condition can be also obtained doing a *von Neumann stability analysis* of this numerical scheme.

The following table summarises the main noteworthy parameters that occurs in this test case.

Test Case n. 1 - Linear Advection Equation	
Param.	Description
Reward Signal	$\|\|u[73:83]\|\|_2$
Neural Network spec.	Mlp - 64x64
x disturbance	x=5
x control	x = 18.2
Nx, space elements	Nx = 200
L length of the domain	L = 50 m
T time domain	T = 0.3s
dx, space discretisation	$dx = L/Nx$
dt, time discretisation	$dt = C \cdot \frac{dx}{c}$
Nt, time steps	$Nt = T/dt$
c, wave propagation speed	$c = 330m/s$

Table 2: Review of Test case n.1 - Linear Advection Equation

3.5 TEST CASE N.2 - BURGERS' EQUATION

Having assessed the capabilities of RL in solving linear problems, the natural next step is a non linear problem.

We here consider the *Burgers equation*, that is a non linear PDE that embodies the key components present in the complete Navier Stokes equations:

- A non linear *advection term*;
- A diffusion term

The experiment carried out was as in test case # 1: the RL agent must control an external perturbation.

3.5.1 *Environment description*

As already stated before, this equation was introduced in order to have a toy model that embodies all the key components of the Navier Stokes equation. It has to be noted that these two terms often conflicts with one another: while non linearity promotes shocks, the diffusion term dumps them out.

The Burgers' equation is expressed as follows,

$$u_{,t} + uu_{,x} = \nu u_{,xx} \tag{40}$$

In the following, Burgers equation will be concisely discussed. For more insights on this influential equation, the reader is referred to the book published by whom firstly formally posed it, J. M. Burgers in [5].

A common procedure when dealing with the fluid dynamics equations is to normalise them, in order to:

- Facilitate the scaling of the results obtained throughout simulations to other flow conditions;
- Avoid computational errors that may arise in dealing with too large/small quantities;
- Assess the relative importance of different terms in the model equation

For these reasons, the non dimensional formulation of Burgers may also be of interest.
Starting from eq. (40):

$$u_{,t} + uu_{,x} = \nu u_{,xx}$$

Each term can be related to a reference:

$$\begin{cases} u = U_{ref}\hat{u} \\ t = T_{ref} \\ x = L_{ref} \\ u_{,x} = \frac{U}{L}\hat{u_x} \\ u_{,t} = \frac{U}{T}\hat{u_t} \\ f(x,t) = F_{ref}f(\hat{x},\hat{t}) \\ u_{,xx} = \frac{U}{L^2}\hat{u_{xx}} \end{cases}$$

Substituting in the original equation, gives:

$$\frac{U_{ref}}{T}\hat{u}_{,\hat{t}} + \frac{U_{ref}^2}{L_{ref}}u\hat{u}_{,\hat{x}} = \nu\frac{U_{ref}}{L_{ref}^2}\hat{u}_{,\hat{x}\hat{x}} + F\hat{f}$$

Then, dividing by U_{ref}/T:

$$\hat{u}_{,\hat{t}} + \underbrace{\frac{U_{ref}^2}{L}\frac{T}{U_{ref}}}_{U=L/T\Rightarrow=1}u\hat{u}_{,\hat{x}} = \nu\frac{T}{\cancel{U_{ref}}}\frac{\cancel{U_{ref}}}{L_{ref}^2} + \frac{FT}{U}\hat{f}$$

This equation can be rearranged as:

$$\hat{u}_{,\hat{t}} + \hat{u}\hat{u}_{,\hat{x}} = \underbrace{\nu\frac{T}{L_{ref}^2}}_{\mathcal{N}} + \underbrace{\frac{FT^2}{L_{ref}}}_{\Phi}\hat{f} \tag{41}$$

Where $\mathcal{N} = \nu\frac{T}{L_{ref}^2}$ and $\Phi = \frac{FT^2}{L}$ represent two dimensionless parameters that will impact the structure of the equation itself. They can be interpreted as a *dimensionless viscosity* and a *dimensionless inertia parameter*, respectively. The effect of such parameters is shown in the next figure.

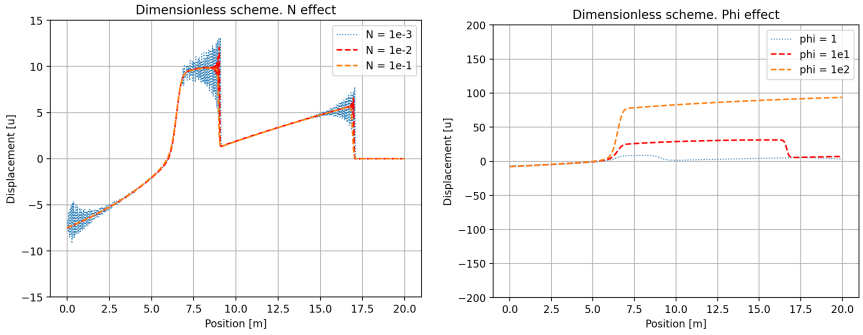

Figure 17: Dimensionless parameters influence

3.5.2 *Implementations*

3.5.2.1 *Explicit Method*

Following the procedure that has been illustrated in the previous test case, the first scheme tested is a finite difference scheme, composed by:

- A *forward derivative* term for the time derivative term;
- A *backward derivative* for the advection term;
- A *centred derivative* approximating the diffusion term at the right hand side.

The full set of the equation and the initial and boundaries condition that will be studied are:

$$\begin{cases} u_{,t} + uu_{,x} = \nu u_{,xx} + f(x,t) \\ \text{i.c.} : u_0 = 0 \\ \text{b.c.} : u_{,x}|_0 = u_{,x}|_L = 0 \end{cases} \tag{42}$$

Hence, from eq. (40) and substituting accordingly, one obtains:

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x} \cdot u_j^n (u_j^n - u_{j-1}^n) + \nu \frac{\Delta t}{\Delta x^2} \cdot (u_{j+1}^n - 2 \cdot u_j^n + u_{j+1}^n)$$

Now, the external perturbation has to be inserted as already done in the Advection test case, so that:

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x} \cdot u_j^n (u_j^n - u_{j-1}^n) + \nu \frac{\Delta t}{\Delta x^2} \cdot (u_{j+1}^n - 2 \cdot u_j^n + u_{j+1}^n) + \Delta t \cdot (f - a) \tag{43}$$

Translating into a Python code gives:

```
#--------------
# Advance simulation one time step
#--------------
u_expl[1:-1] = u1_expl[1:-1] + (dt / dx) * u1_expl[1:-1] *
            (u1_expl[1:-1] - u1_expl[:-2]) \
            + nu * (dt / (dx ** 2)) *
            (u1_expl[2:] - 2 * u1_expl[1:-1] + u1_expl[:-2])
            + dt * (fun[1:-1] + action_vec[1:-1])

# Boundary Conditions
u_expl[0] = u1_expl[1]
u_expl[-1] = u1_expl[-2]

# Switch variables before next step
self.u3[:], self.u2[:], self.u1[:] = self.u2, self.u1, self.u
```

Python Code 9: Burgers equation time stepping

3.5.2.2 *Crank Nicolson Implicit Method*

Recalling briefly the discretized Burgers equation:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + \frac{u_j^n(u_j^n - u_{j-1}^n)}{\Delta x} - \nu\frac{(u_{j-1}^n - 2u_j^n + u_{j+1}^n)}{\Delta x^2}$$

For the moment the force/action term will be neglected. It will be added as a final step in the implicit system.

Writing the equation in the *conservative form*:

$$\frac{\partial u}{\partial t} + \frac{\partial F}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{44}$$

with the flux F being:

$$F = \frac{u^2}{2} \tag{45}$$

Substituting in the discretized equation, then

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + \frac{F_j^n - F_{j-1}^n}{\Delta x} - \frac{\nu(u_{j-1}^n - 2u_j^n + u_{j+1}^n}{\Delta x^2} = 0 \tag{46}$$

A Crank - Nicolson implicit formulation can be expressed as follows:

$$\frac{\Delta u_k^{n+1}}{\Delta t} = -\frac{L_x(F_j^n + F_j^{n+1})}{2} + \nu\frac{L_{xx}(u_j^n + u_j^{n+1})}{2} \tag{47}$$

with

$$\begin{cases} \Delta u_j^{n+1} = u_j^{n+1} - u_j^n \\ L_x = \frac{(-1,0,1)}{\Delta x} \\ L_{xx} = \frac{(1,-2,1)}{\Delta x^2} \end{cases}$$

The most efficient way to deal with such a system would be solving tridiagonal equations for computing the solution. Hence, using a Taylor expansion and linearising with Δt, gives:

$$F_j^{n+1} = F_j^n + \Delta t(\frac{\partial F}{\partial t})_j^n + \frac{\Delta t^2}{2}(\frac{\partial^2 F}{\partial t^2})_j^n + ... \tag{48}$$

Substituting

$$A = (\frac{\partial F}{\partial u})_j^n = u_j^n \tag{49}$$

Then the tridiagonal form is

$$u_j^{n+1} + \frac{1}{2}\Delta t[L_x(u_j^n u_j^{n+1}) - \nu L_{xx}u_j^{n+1}] = u_j^n + \frac{1}{2}\nu\Delta t L_{xx}u_j^n \tag{50}$$

Concisely:

$$a_j^n u_{j-1}^{n+1} + b_j^n u_j^{n+1} + c_j^n u_{j+1}^{n+1} = d_j^n \tag{51}$$

Where

$$\begin{cases} a_j^n = \frac{\Delta t}{4\Delta x} u_{j-1}^n - \frac{s}{2} \\ b_j^n = 1 + s \\ c_j^n = \frac{\Delta t}{4\Delta x} u_{j+1}^n - \frac{s}{2} \\ d_j^n = 0.5 \cdot s \cdot u_{j-1}^n + (1-s) u_j^n + \frac{s}{2} u_{j+1}^n \\ s = \frac{\nu \Delta t}{\Delta x^2} \end{cases} \tag{52}$$

That is *unconditionally stable* in the Von Neumann analysis sense and has a truncation error of $\mathbf{O}(\Delta t^2, \Delta x^2)$. Turning all this procedure into Python code:

```
#--------------
# Advance simulation one time step
#--------------
for cont in range(Nt):
# --- DIMENSIONAL IMPLICIT BURGERS
    s = (nu * dt) / (dx ** 2)
    a = -(dt / (4 * dx)) * u1[:-2] - s / 2
    a = np.concatenate((a, - (dt/(4*dx)) * u1[-2] - 0.5 * s), axis = None
    )
    b = (1 + s) * np.ones(Nx - 2)
    b = np.concatenate((- (dt/(4 * dx)) * u1[0] + 0.5 * s + 1, b),axis =
    None)
    b = np.concatenate((b, (dt/(4 * dx))*u1[-1] + 0.5 * s + 1),axis =
    None)
    c = (dt / (4 * dx)) * u1[2:] - s / 2
    c = np.concatenate(((dt/(4*dx)) * u1[1]  - s/2, c), axis = None)
    d = 0.5 * s * u1[:-2] + (1 - s) * u1[1:-1] + (s / 2) * u1[2:]
    d = np.concatenate((d, 0.5 * s * u1[-2] + u1[-1]*((1 - s) + 0.5 * s))
    , axis=None)
    d = np.concatenate(( u1[0]*((1 - s) + 0.5 * s) + 0.5* s * u1[1] , d),
     axis=None)
    k = np.array([a, b, c])
    offset = [-1, 0, 1]
    A = diags(k, offset).toarray()
    # RHS terms
    forcing = AMPLITUDE * (np.sin(2 * np.pi * FREQUENCY * t[kk]))
    fun = forcing * np.exp(-((x - XF) ** 2) / (2 * SIGMA ** 2))
    b = d + dt*fun
    # Solving for timestep
    u = np.linalg.solve(A, b)
    u1 = u

    # Switch variables before next step
    self.u3[:], self.u2[:], self.u1[:] = self.u2, self.u1, self.u
```

Python Code 10: Burgers implicit method

Where the Neumann boundary condition this time is embodied in the first and last equation, respectively. For instance, considering the above eq. (51) in the first node:

$$a_0^n u_{-1}^{n+1} + b_0^n u_0^{n+1} + c_0^n u_1^{n+1} = d_0^n \tag{53}$$

But u_{-1}^n is outside the computational domain. In order to avoid this problem a *ghost node* is introduced. Essentially, considering a backward derivative at $x = 0$ one can write:

$$\frac{du}{dx}|_0 = \frac{u_0 - u_{-1}}{\Delta x} \tag{54}$$

But the Neumann condition that is imposed prescribes that:

$$\frac{du}{dt}|_0 = \frac{du}{dt}|_L = 0 \tag{55}$$

Hence,

$$u_0 = u_{-1} \tag{56}$$

Substituting this relation into eq. (53), the problem of the computational domain is avoided, coming to:

$$a_0^n u_0^{n+1} + b_0^n u_0^{n+1} + c_0^n u_1^{n+1} = d_0^n \tag{57}$$

Arranging the terms,

$$u_0^{n+1} \underbrace{(a_0^n + b_0^n)}_{\mathbf{B}[0]} + \overbrace{c_0^n}^{\mathbf{C}[0]} u_1^{n+1} = \underbrace{d_0^n}_{\mathbf{D}[0]} \tag{58}$$

For the other extreme of the boundary, at $x = L$ the same passages lead to:

$$\underbrace{a_N^n}_{\mathbf{A}[-1]} u_{N-1}^{n+1} + u_N^n \overbrace{(b_N^n + c_N^n)}^{\mathbf{B}[-1]} = \underbrace{d_N^n}_{\mathbf{D}[-1]} \tag{59}$$

3.5.2.3 *Validation of numerical methods via Cole-Hopf transformation*

In order to validate the proposed numerical schemes, the theoretically exact solution via *Cole - Hopf transformed diffusion equation* is used. This transformation transforms the nonlinear Burger's equation into a linear parabolic one. Hereafter a brief overview on how this transformation is achieved is given. For a more detailed discussion, the reader is referred to [26].

Considering the Burger's equation as an initial value problem:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \tag{60}$$

Given an I.C.:

$$u(x,0) = u_0(x) \tag{61}$$

Eq. 60 can be linearised considering the Cole-Hopf transformation:

$$u(x,t) = -\frac{2\nu\phi_x}{\phi} \tag{62}$$

Substituting,

$$u = \Psi_x \tag{63}$$

Then,

$$\Psi_{xt} + \Psi_x\Psi_{xx} = \nu\Psi_{xxx} \tag{64}$$

Using the chain rule:

$$\Psi_{xt} + \frac{\partial}{\partial x}\left(\frac{1}{2}\Psi_x^2\right) = \nu\Psi_{xxx} \tag{65}$$

Integrating with respect to x yields:

$$\Psi_t + \frac{1}{2}\Psi_x^2 = \nu\Psi_{xx} \tag{66}$$

Substituting in eq. 66:

$$\Psi = -2\nu \ln\phi \tag{67}$$

Leads to

$$\phi_t = \nu\phi_{xx} \tag{68}$$

That is the *heat equation*.
At $t = 0$, solving eq. (62) for ϕ gives:

$$\phi(x,0) = Ce^{-\frac{1}{2\nu}\int u_0\,dx} \tag{69}$$

Where C is a constant that has no influence on the final solution. Finally, using:

$$u_0(x) = \sin x$$

And imposing a Neumann boundary condition such that:

$$u(0,t) = 0 = u(2\pi,t)$$

The initial-boundary value problem of a heat equation is obtained:

$$\begin{cases} \phi_t = \nu\phi_{xx} \\ \text{i.c.}: \ \phi(x,0) = e^{\frac{\cos x}{2\nu}} \\ \text{b.c.}: \ \phi_x(0,t) = 0 = \phi_x(2\pi,t) \end{cases} \tag{70}$$

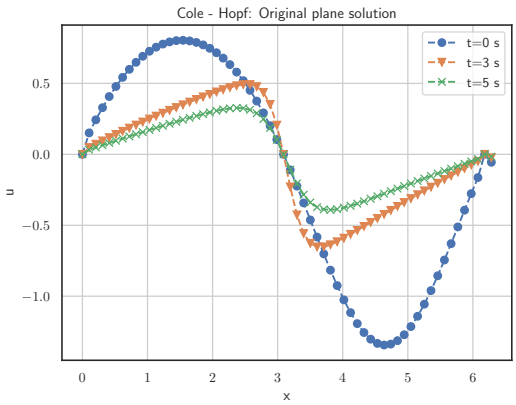

Figure 18: Exact solution of the transformed Burger's equation

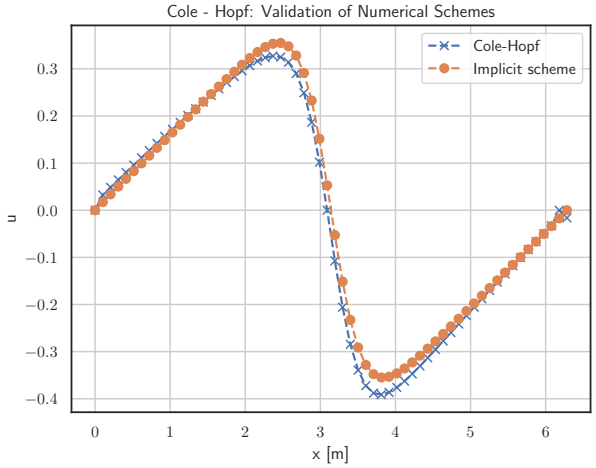

Figure 19: Validation of the numerical schemes for Burger's equation via Cole-Hopf transformation

Which gives to the exact solution of the Burger's equation.

Hence, these solutions can be compared with the numerical scheme proposed above in this section.

As it can be seen from Fig. 19 the implicit scheme is very accurate and almost superpose itself with the analytical solution. However, looking at the explicit scheme there is a significant difference, caused by *numerical diffusion*.

3.5.2.4 *Time discretisation*

Even if the proposed scheme has been above validated, another concern about the stability of this scheme is the *time - discretisation step*. Recalling the Advection test case, in eq. 37 was discussed the necessary condition (but not sufficient) to satisfy in order to have a stable numerical scheme. But in this case the wave propagation speed (called c in the Advection analysis) is no longer constant. Following

A. Quarteroni in [23], one can take up the previous CFL condition and changing it as

$$\Delta t \leqslant \frac{\Delta x}{\sup_{x \in \mathbf{R}} |c(x,t)|} \tag{71}$$

Doing so, essentially the CFL condition is computed by assuming the maximum velocity in the whole field. It has to be noted that the very origin of non linearity is due to the wave propagation speed that becomes a function of x. If this condition is satisfied, it can be shown again that the Upwind numerical scheme is strongly stable in norm $\|\cdot\|_1$.

As a consequence then, the *time - discretisation step* should be studied for every particular set of the computational domain and for each boundary or initial condition applied, since with these parameters also the maximum wave propagation may change. In order to be free of such a study but without renouncing on the stability of the numerical scheme, the *implicit method* is used.

3.5.2.5 *Computational complexities of proposed schemes*

Even if it does guarantee stability, the implicit method implies a greater computational weight, requiring the solution of a system for each timestep. In particular, given a IV (initial value) problem,

$$\begin{cases} \dot{x} = f(x,t) \\ x(0) = x_0 \qquad x \in \mathcal{R}^N \end{cases} \tag{72}$$

the complexity for an *explicit solver* is the same of evaluating $f(x,t)$. For the sake of generality, assuming that $f(x,t)$ depends on all the x elements, it results $\mathcal{O} \sim (N^2)$, where N is the dimension of the computational domain. This big-O complexity is intended per *each timestep*.

On the other hand, implicit methods solve a linear system of equation for each timestep. This process gives birth to a matrix of dimensions $(N \times N)$. Solving such a linear system in the most naive way, e.g.: using Gaussian elimination, is $\mathcal{O} \sim (N^3)$ complex, per *each timestep*.

Finally, it is noted how this complexity can easily become more restrictive than the effective time step imposed in the simulation, becoming then the effective complexity of the step.

3.5.2.6 *Exploiting the structure of tridiagonal systems*

Motivated by the discussion in Sec. 3.5.2.5, a more efficient way to solve the tridiagonal system is searched.

Dealing with an implicit method to solve the Burgers equation for each time step, the computational cost skyrockets with even limited

computational domains. Hence, summing up this computational cost with the one implied in using RL algorithms, there would be a major slowdown that could compromise the wall time required by the simulation. In order to face this issue, the tridiagonal structure of the matrix has to be exploited using some more efficient algorithm.

As a first approach *Thomas algorithm* is tested.
This algorithm is used to solve *tridiagonal* systems since it is proved to retain the exact solution then requiring only $\mathcal{O}(N)$ operations instead of the $\mathcal{O}(N^3)$ for the classic Gaussian approach. However, this cannot be always used since it is guarantee to be stable only if:

- The matrix is diagonally dominant (either by rows or columns)
- The matrix is symmetric positive definite

Generally a tridiagonal system of n unknowns may be written as:

$$\begin{bmatrix} b_1 & c_1 & \dots & \dots & \dots & \dots & 0 \\ a_2 & b_2 & c_2 & & & & \vdots \\ 0 & a_3 & b_3 & c_3 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & c_{n-1} \\ 0 & \dots & \dots & \dots & \dots & a_n & b_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ \vdots \\ d_n \end{bmatrix} \tag{73}$$

We now compute the new coefficients for the computation, that will be denoted with primes:

$$c_i' = \begin{cases} \frac{c_i}{b_i} & \text{for } i = 1 \\ \frac{c_i}{b_i - a_i c_{i-1}'} & \text{for } i = 2, 3, ..., n-1 \end{cases} \tag{74}$$

and

$$d_i' = \begin{cases} \frac{d_i}{b_i} & \text{for } i = 1 \\ \frac{d_i - a_i d_{i-1}'}{b_i - a_i c_{i-1}'} & \text{for } i = 2, 3, ..., n \end{cases} \tag{75}$$

Then the solution can be obtained as

$$x_n = d_n' \tag{76}$$

and

$$x_i = d_i - c_i x_{i+1} \quad \text{for } i = n-1, n-2, ..., 1 \tag{77}$$

Therefore, the algorithm may be concisely written as in algorithm 5:

In Python:

Algorithm 5: Thomas Algorithm

for $i = 2,\dots, n$ **do**

$w = \frac{a_i}{b_{i-1}}$;

$b_i := b_i - w \cdot c_{i-1}$;

$d_i := d_i - w \cdot d_{i-1}$;

$x_n = \frac{d_n}{b_n}$;

$x_i = \frac{d_i - c_i \cdot x_{i+1}}{b_i}$;

end

```
def Thomas_solver(a, b, c, d):
    '''
    Thomas algorithm implementation in Python.
    '''
    nf = len(d)  # number of equations
    ac, bc, cc, dc = (x.astype(float) for x in (a, b, c, d))
    for it in range(1, nf):
        mc = ac[it - 1] / bc[it - 1]
        bc[it] = bc[it] - mc * cc[it - 1]
        dc[it] = dc[it] - mc * dc[it - 1]

    xc = bc
    xc[-1] = dc[-1] / bc[-1]

    for il in range(nf - 2, -1, -1):
        xc[il] = (dc[il] - cc[il] * xc[il + 1]) / bc[il]

    return xc
```

Python Code 11: Thomas algorithm

As an alternative to such a method, also *SciKit Solve Banded* is tried [16]. To do so, the starting matrix has to be rearranged in a more compact way:

$$\begin{bmatrix} b_1 & c_1 & \dots & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & \ddots & \ddots & \ddots \\ 0 & \dots & a_n & b_n \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & b_1 & c_1 \\ a_1 & b_2 & c_2 \\ \vdots & \vdots & \vdots \\ a_{n-1} & b_n & 0 \end{bmatrix}$$

From a computational viewpoint, dealing with such matrices is always more advisable with respect to the full dimensioned matrices. The complexity of performing operations, such as multiplications for instance, is greatly reduced.

In Python:

```
N = len(b)
Ab = np.zeros((3, N))
Ab[0, 1:] = c
Ab[1, :] = b
```

```
Ab[2, :-1] = a
```

Python Code 12: From a tridiagonal matrix to a band one

Where a, b and c are the three diagonals of the tridiagonal matrix respectively.

Finally, in order to make a comparison fair and robust between the methods here proposed, a random matrix of different dimensions `Ns=[100, 500, 1000, 2000, 10000, 20000, 100000]` has been used for the test, and each function has been tried, imposing `n_trials=100`. The results of this test are the following: Each method is represented

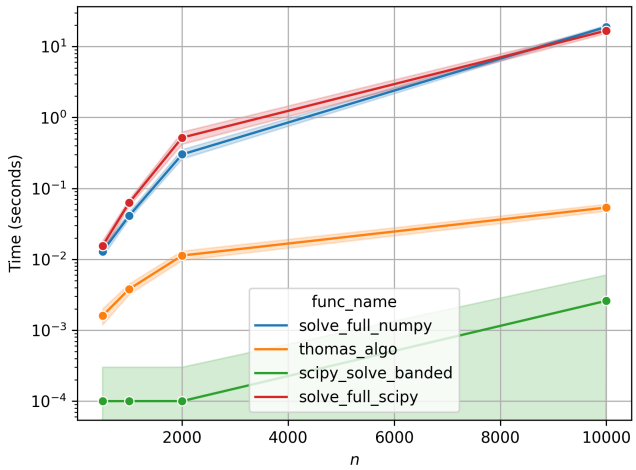

Figure 20: Benchmark of tridiagonal matrix solving methods

by a trend and the area coloured of the same colour is the area subtended by the fastest and the slowest case for this method across the 100 tests conducted. Hence, `Solve Banded` will be the method of choice in this test case implementation.

3.6 TEST CASE N.3 - VON KARMAN VORTEX STREET AFTER A CYLINDER

This test case has been published from J. Rabault et al. in [24] in which a deep reinforcement learning agent has been made to interact with a fluid flow computed through a step-by-step CFD simulation. This test case stands for the leading interest in Fluid Mechanics in these active flow control techniques.

This test case will not be solved in this work - since it has already been studied deeply be the research team that defined this problem. Instead, the focus in looking at this test case will be more didactic, trying to understand the control law chose by the DRL agent.

Thereafter the main points of the experiment are recalled, but for more information see the original paper.

3.6.1 *Environment Description*

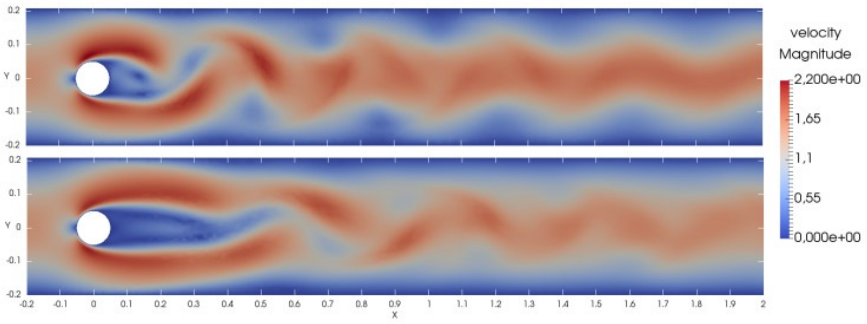

Figure 21: Velocity magnitude illustrating the effect of flow control

In this environment, a 2D simulation of the non-dimensionalised flow around a cylinder is described by incompressible Navier-Stokes equations at $Re = 100$. The action of the agent is embodied by two jets of angular width of 10° which are set on the sides of the cylinder and inject fluid in the direction normal to the cylinder surface, directly influencing the flow field.

These jets are controlled through their flow rates Q_1 and Q_2, and a constraint is imposed in order to avoid net mass flow rate to be introduced into the system:

$$Q_1 + Q_2 = 0 \tag{78}$$

In this work the objective was to minimise the drag D through a reduction of the vortex shedding past the cylinder. Hence, the *reward* is defined as:

$$r = \bar{C_{DT}} - |\bar{C_{LT}}| \tag{79}$$

that are the mean over one full vortex shedding cycle.

Finally, the results are expressed plotting the normalised value of the drag:

$$C_{D,0} = \frac{D}{\rho \bar{U}^2 R} \tag{80}$$

where $\bar{U} = 2U(0)/3$ is the mean velocity magnitude, ρ is the volumetric mass density of the fluid and R the diameter of the cylinder. $U(0)$ is the velocity profile at the inflow. Also the mass flow rates of the two cylinders are normalised,

$$Q_i^* = Q_i/Q_c \tag{81}$$

where

$$Q_c = \int_{-R}^{R} \rho U(y) dy \tag{82}$$

is the mass flow rate introduced by the inlet profile that intersects the cylinder diameter.

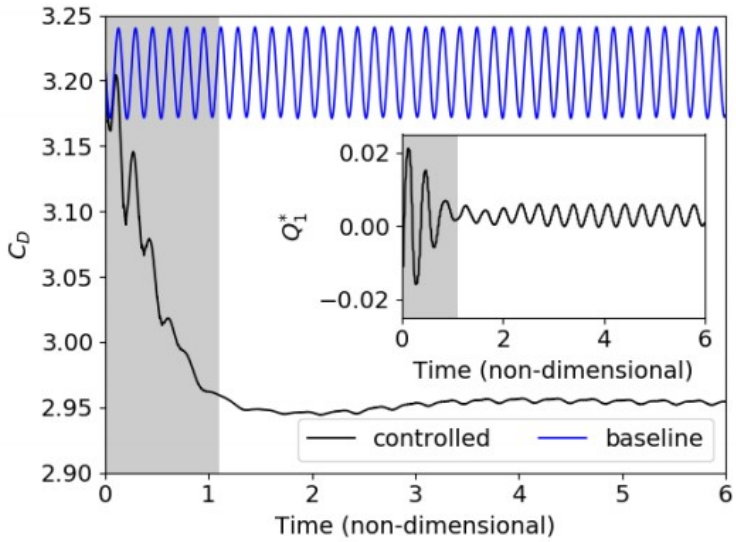

Figure 22: Results in J. Rabault et al. [24]

The results obtained by the research team applying this DRL controller are the following: after around 120 training epochs (number of updates on the trajectories), the drag coefficient is reduced by around 8% while the fluctuations in lift are reduced by around 75%. Finally, the area A of the recirculating bubble is drastically increased by 125%.

3.6.2 *Implementation*

Since the agent is interacting with a fluid flow field a CFD solver is needed. The authors chose the open source Python package FEniCS [25].

For what concerns the DRL framework they used Tensoforce and TensorFlow, see M. Abadi et al. [1].

The main simulation's parameters are recalled hereafter.

Test Case n. 3 - von Karman vortex street

Param.	Value
R	0.05
L	2.2
H	0.41
ν	10^{-3}
ρ	1
Re	100
dt	$dt = 5 \cdot 10^{-4}$
θ_1, jet 1	90°
θ_2, jet 2	270°
γ, jets width	10°

Table 3: Review of Test case n.3 - DRL control of the 2D von Karman vortex street

Finally, in order to help *PPO* learn the correct set of continuous control signal, two limitations have been applied:

i The control provided by the agent is held constant for a duration of roughly 10% of the vortex shedding period;

ii The control is made continuous in time. To this end, the control is obtained for each jet as:$c_{s+1} = c_s + \alpha(a - c_s)$, where c_s is the control of the jet at the previous time step and a is the action provided by the PPO agent. $\alpha = 0.1$ is a numerical parameter.

4

RESULTS

RESULTS ORGANISATION

In this Section, the Results obtained on the three test cases will be presented. The organisation for each is the following: at first the *neural network architecture (NNA)* for both the policy and value network is studied. This study comprehends tests in the structure of the Neural Network (NN) along tests on its activation function. One parameter is varied at time, using as default settings - $(64, 64)$ arcitecture, ReLU - for all the others. Three Multilayer Perceptron (MLP) are investigated, $(64, 64)$, $(150, 50, 25)$ and $(400, 300)$. For what concerns the activation functions, they are varied across tanh, ReLU and Leaky ReLU. These choices are made following commonplace trends in the community. Also the performances of MLPLSTM networks are evaluated. Secondly, the architecture with the highest performance will be studied more in depth, giving an insight also on the main hyper parameters that come into play. After this step, an Hyper Parameter Optimisation (HPO) is carried out through Bayesian Optimisation (BO) to get the best performance out of the Deep Reinforcement Learning agent. Being this DRL-study completed, this performances are compared with more classical optimisation techniques - for instance Nelder-Mead's simplex method - and with Bayesian Optimisation, used here with the purpose of control.
In the study of test case #3: the control of a von Kàrmàn vortex street after a cylinder, the DRL-study of above is omitted since it is already present in the published work of its authors in J. Rabaud et al. [24].

4.1 ADVECTION EQUATION

The physical set-up of the experiment is the following:

Simulation settings

	description	value
L[m]	length of the physical domain	50
T[s]	simulation duration	0.3
Nx[-]	space elements	(2e2)
dx[-]	space discretisation step	L/Nx
c[m/s]	wave propagation speed	330
dt[-]	time discretisation step	$C \cdot dx/c$
Nt[-]	time elements	T/dt
f[Hz]	perturbation's frequency	0.5e2
A[-]	perturbation's amplitude	3e2
C[-]	Courant number	1

Table 4: Advection simulation setting

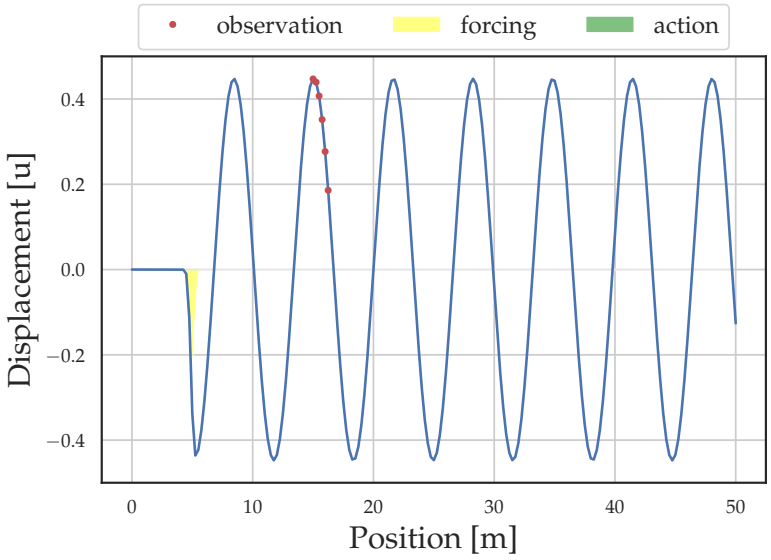

Figure 23: Advection Equation environment. Overview without any control.

The *Deep Reinforcement Learning* set-up is:

Deep Reinforcement Learning settings

Method	seeds	$\mathcal{A}$	$\mathcal{O}$	reward shape r
PPO	4	$(1,)$	$(18,)$	$-\|u\|_2$

Table 5: DRL agent settings

where $\mathcal{A}$ and $\mathcal{O}$ represent the action space and the observations space, using `Python` notation, respectively.

4.1.1 *Neural Network Architecture (NNA)*

As a first step in studying the performances of the DRL agent in controlling the wave perturbation, the architecture of the agent itself has to be chosen. In particular, this implies a study of the *neural network structure* and *activation function*. These tests are conducted using a fixed number of steps (20M) for the agent to interact with the environment. The evaluation metrics are: the mean reward obtained by the agent after learning in 100 episodes, the dispersion (σ) using different seeds, where

$$\sigma^* = \left| \frac{(max - min)}{(max)} \right| \tag{83}$$

Where *max* and *min* are the best and the worst performance respectively, for a given seed.

4.1.1.1 *Architecture*

Advection: Neural Network Architecture

	$(150, 50, 25)$	$(400, 300)$	$(64, 64)$
$\bar{r}_t$	-9.324	-11.667	-8.245
σ	0.380	0.462	0.271

Table 6: Advection:NNA study

Table 6 shows that $(64, 64)$ architecture not only achieves a better reward across 100 episodes, but it is also less sensitive to initial seeds. Moreover, the number of weights to be optimised influences the (temporal) length of the training, for a given number of time steps. However, a metric about time is not presented here since different desktop computers were used, and the training time also depends on the performance of the machine in use.

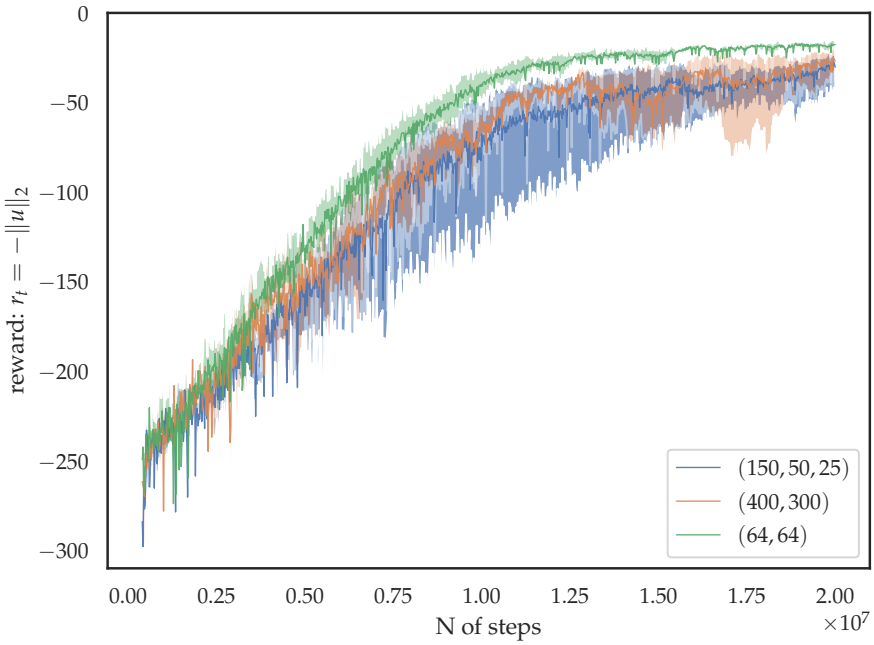

Figure 24: Advection: Behaviour of the collected reward for different Neural Network Architectures

Looking at both Fig. 24 and Fig. 27 it can be appreciated how the architecture $(150, 50, 25)$ is the most sensitive to the initial random seed.

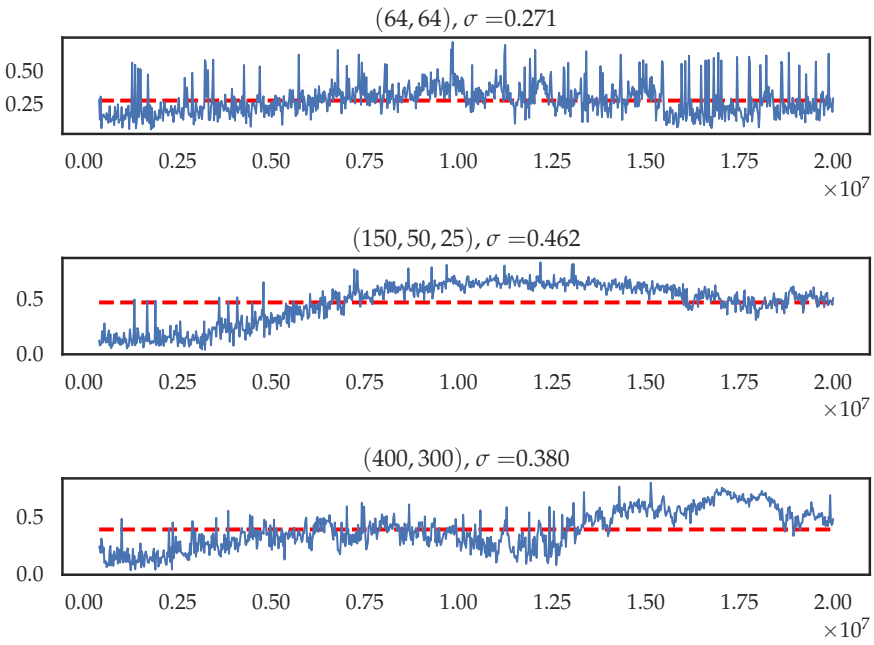

Figure 25: Advection: Detail on dispersion for a given architecture using different initial seeds. Plot of σ per each time step.

4.1.1.2 *Activation Function*

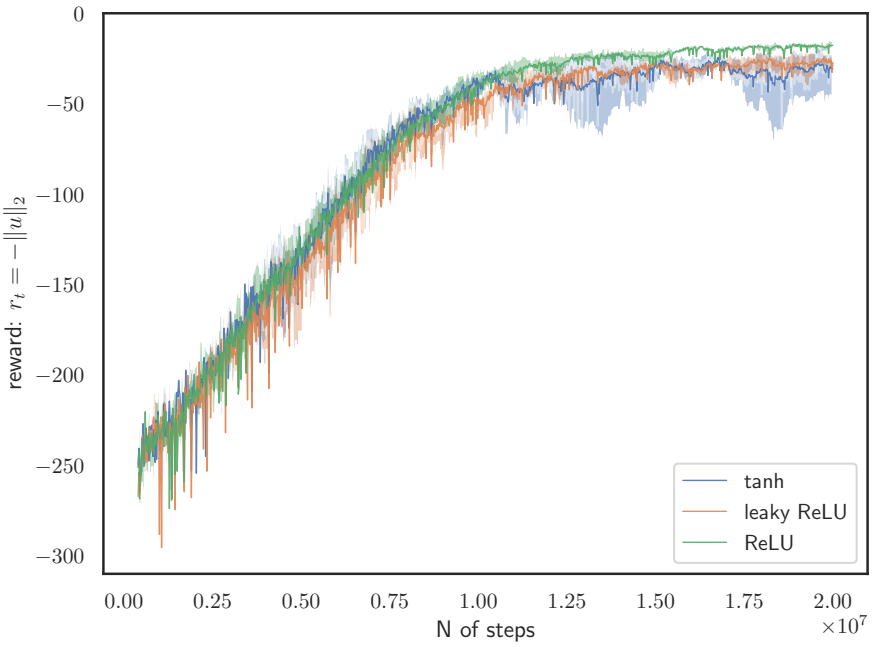

Figure 26: Advection: Behaviour of the collected reward for different Activation Functions

Advection: Neural Network Architecture

	tanh	leaky ReLU	ReLU
$\bar{r}_t$	−11.900	−11.566	−8.245
σ	0.390	0.306	0.271

Table 7: Advection: reward and dispersion as a function of the Activation Function

From the results in Tab. 8, ReLU is the activation function that both limits dispersion and achieve a best overall reward over 100 episode of test.

4.1.1.3 *Long Short Term Memory (LSTM)*

As a final test in this architecture study, the nature of neurons themselves that compose the net is changed. Long Short Term Memory are used, and compared with the best MLP originated by the previous study: $(64, 64)$ with ReLU activation function.

The observation about computational time made in Sect. 4.1.1.1 is especially true in using LSTM networks. In fact, the additional com-

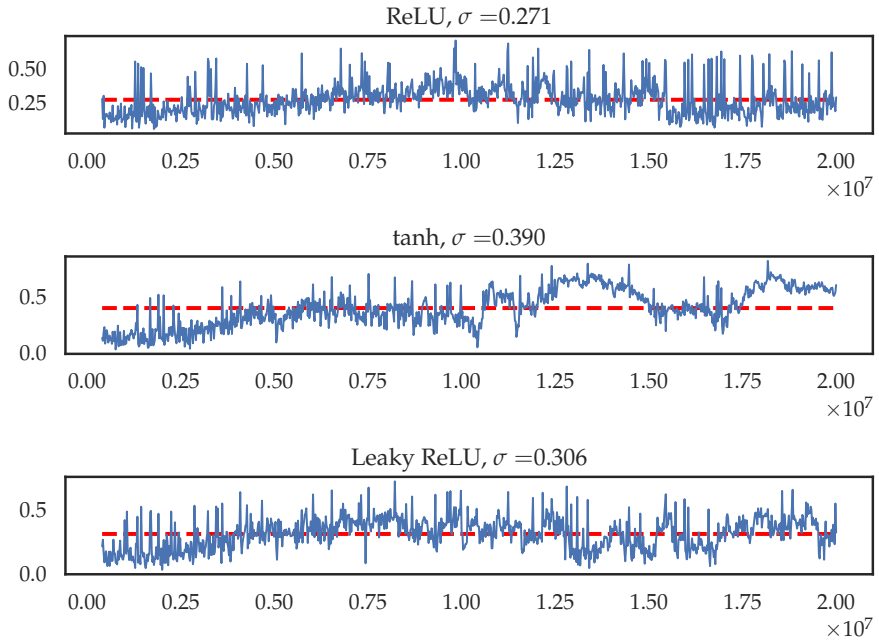

Figure 27: Advection: Activation Function, detail on dispersion for a given architecture using different initial seeds. Plot of σ per each time step.

Advection: MLP against MLPLSTM

	MLPLSTM	MLP
$\bar{r}_t$	−12.011	−8.245
σ	0.306	0.271

Table 8: Advection: Long Short Term Memory performances

putations greatly slow down the simulation that requires almost double the time of a standard MLP[1].

Even though the MLPLSTM-agent is able to successfully control the wave and to achieve a reward not so distant from the MLP-agent's one, the importance of dispersion with different seeds is of major interest. In fact, important oscillations in the collected rewards are present, as $\sigma = 0.706$ does testify. It is noted that, because of these oscillations, the plot of Fig. 28 is rearranged as follows: instead of plotting the mean and to fill between the best and the worst performance for each time step, the coloured area represents the distance between the maximum and the average performance. Being the minima too low, it would have impacted the readability of the plot.

1 Keeping in mind that these performances are also desktop-dependent.

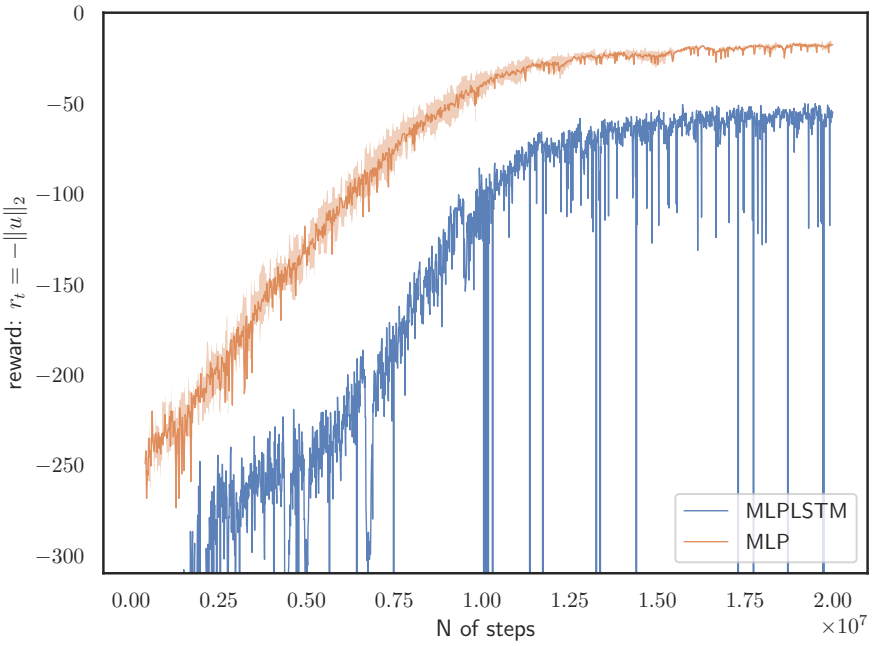

Figure 28: Advection: MLPLSTM performances

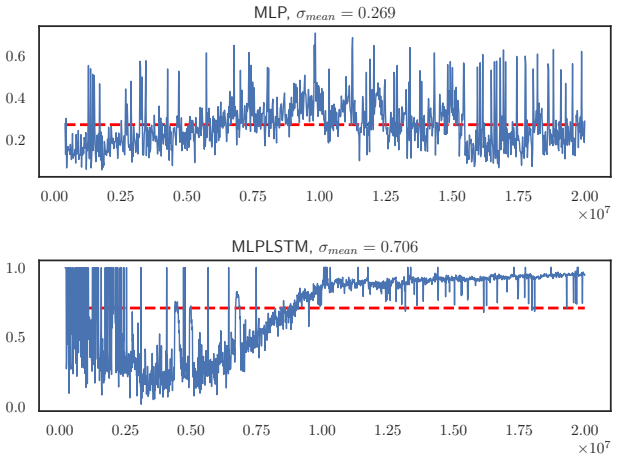

Figure 29: Advection: MLPLSTM, detail on dispersion for a given architecture using different initial seeds. Plot of σ per each time step.

Hence, this complexity added on top of "classic" MLP not improves the performances of the agent as hoped, at least in this environment.

4.1.2 *DRL Control Performances*

A more in-depth analysis of the control action made by the agent, exploiting the best architecture, can be made.

Hyper parameters for this simulation were empirically chosen and are reported below in Tab. 12.

Main Hyperparameters choices

Method	n. of steps	n. minibatches	noptepochs	learning rate	cliprange
PPO	$3.2 \cdot 10^4$	100	5	$2.5 \cdot 10^{-4}$	0.2

Table 9: Naive PPO simulation parameters

The agent successfully learned an effective control action relying solely on the information obtained interacting with the environment. Such information were passed by the observation points, as explained in Sect. 3.4. As it can be seen looking at the displacement field, the external perturbation is almost cancelled out.

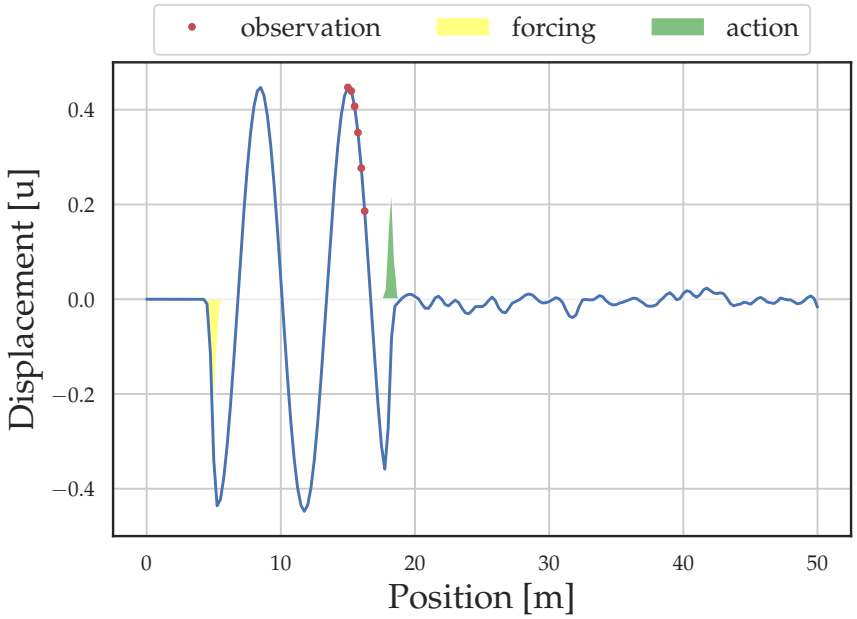

Figure 30: Advection: Controlled wave

Moreover, this control law can be compared with the theoretical one, being formally known:

$$g(x,t) = A \sin \omega t - \omega \frac{\Delta x}{c} + \pi \cdot \mathcal{N}(x - (5 + \Delta x), 0.2)$$

Hence, looking at Fig. 31 it can be seen how the control applied by the DRL agent and the theoretical solution are almost superposing. A

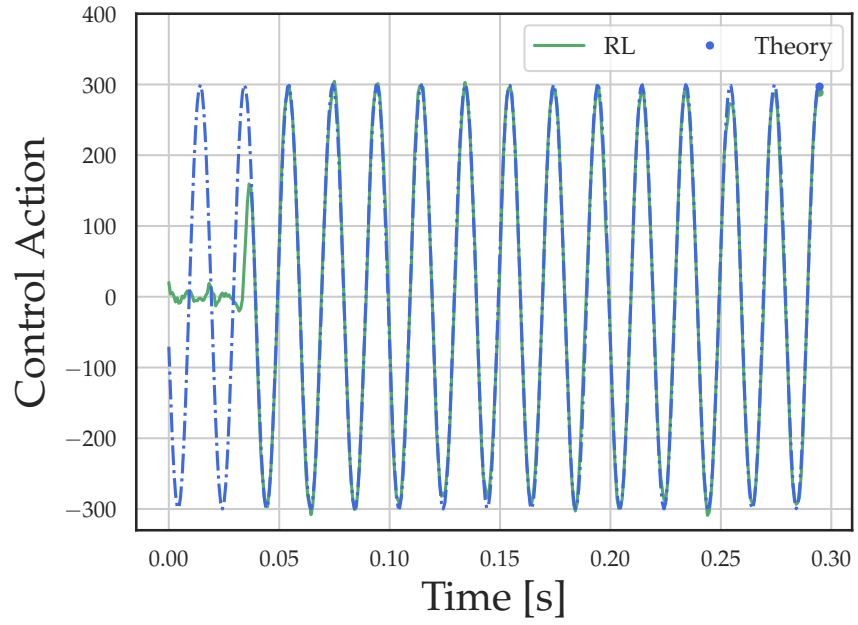

Figure 31: RL Control Action against the theoretical one.

confirm of that can be searched in the plane of frequencies, using a *Fourier Transform* of the control action.

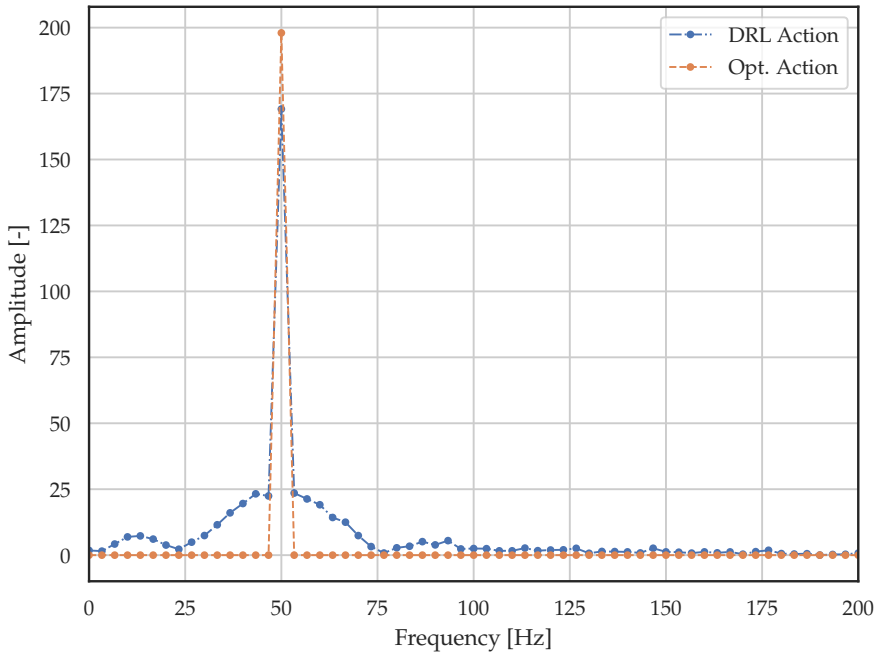

Figure 32: Advection: Fourier Transform of the control action

The highest frequency among the DRL-control amplitudes is the same of the exact solution at $f = 50$ [Hz]. The same frequency which perturbs the system. However, the DRL appears to be more *noisy*,

due to its exploration and to the fact that the control law is obtained as a sequence of trials and errors. *How much of the control power is wasted in these lower frequencies?*. In order to answer to this question, the concept of *Power Spectral Density (PSD)* is introduced. Essentially, it should explain how the power is distributed across the frequency.

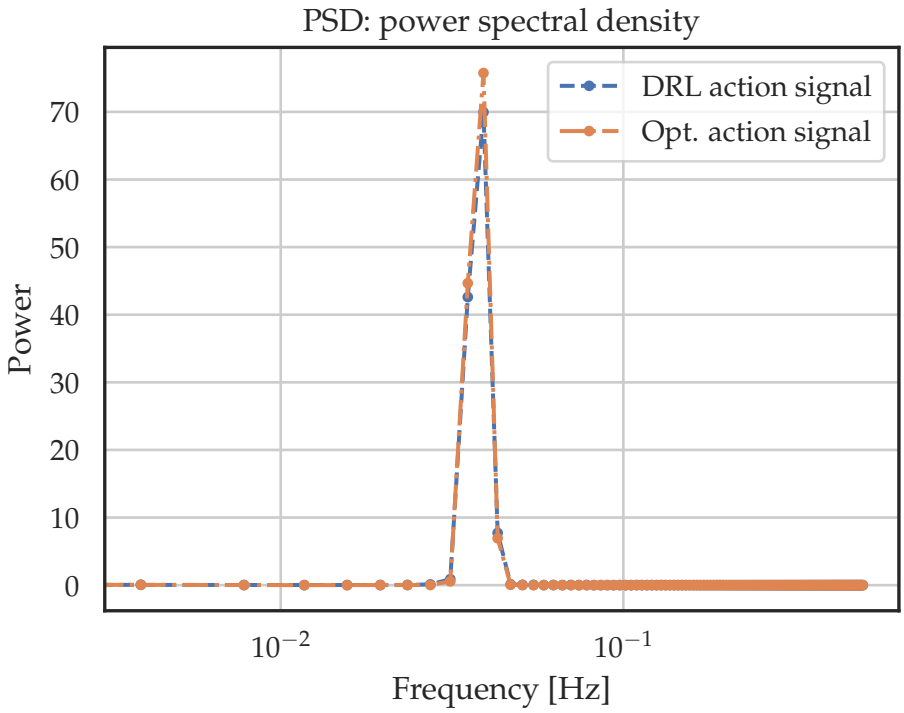

Figure 33: Advection: Power Spectral Density (PSD)

The power distribution of the DRL-control and the theoretical are almost identical, that means that even if some other frequencies are present in the Fourier transform of the control action, not a lot of energy is employed therein.

4.1.3 *Improving the learning performances: Hyper parameters optimisation*

Now that a baseline of the DRL agent is obtained, an optimisation operation is carried out with the scope of improving the overall performances of the DRL agent. The *hyper parameter space* is defined as in Tab. 10.

This ranges have been retrieved consulting main PPO published papers and specialistic publication, see for a reference [32]. The hyper parameter space is hence 8-th dimensional.

Bayesian Optimisation technique has been chosen for the completion of this task, in particular the Optuna Python library ([2]). The optimisation process is carried out as follows: at first the optimiser suggests

Hyper parameter space

Hyper parameter	boundaries	type
n. of steps	[8, 32 000]	integer
discount factor (γ)	[0.8, 0.9997]	logarithmic
learning rate (α)	[1e-5, 1.]	logarithmic
entropy coefficient	[5e-6, 3e-3]	logarithmic
cliprange	[0.1, 0.4]	float
noptepochs	[3, 30]	integer
lam	[0.9, 1]	float
value function coeff.	[0.5, 1]	logarithmic

Table 10: PPO Hyper parameter space

a set of values, these values are used to define an agent that interacts with the environment. Using the callback described in Sect. 3.1.1 if the standard deviation of collected rewards across different episodes is stable under a 15% range, the simulation is ended.

It took the optimiser *86 trials* - i.e.: 86 simulations - to obtain the best combination of hyper parameters for this environment.

This study also gives insights about the relative importance of the different hyper parameters on the agents' performance. The importances are represented by floating point numbers that sum to 1.0 over the entire set. The higher the value, the more important is the hyper parameter.

It can be seen from Fig. 34 how the *learning rate* is the most important hyper parameter by far, followed by the *value function coefficient* and by the *number of steps*. The importance of the learning rate does not surprise: it defines the gradient descent update stepping, always a crucial parameter, even more when dealing with on-policy methods, such as PPO. It has to be stressed, however, how these results are strictly linked to the problem at stake and to the method employed.

Finally, a direct comparison between the so-called *naive agent* - i.e.: the not optimised - and the optimised one can be carried out, comparing the hyper parameters value and the effectiveness of the obtained best policy.

In Fig. 36 a learning curve of the two agents is proposed.

Hence, the optimal agent learns a robust policy interacting with the environment for 455k timesteps with a much steeper learning curve slope. Moreover, the sensitivity from the initial seed is extremely close to the naive one, as Fig. 37 shows.

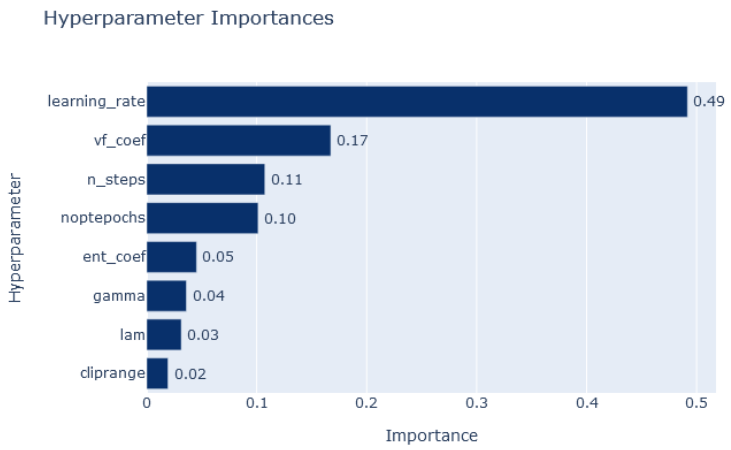

Figure 34: Advection hyper parameters importances

Hyper parameters comparison

Hyper parameter	Naive	Optimised
gamma	9.90E-01	9.84E-01
n. of steps	3.20E+04	3.33E+02
learning rate	2.50E-04	1.05E-04
vf. coef	5.00E-01	5.26E-01
ent. coef	1.00E-02	1.49E-04
noptepochs	5.00E+00	2.22E+01
lam	9.50E-01	9.82E-01
cliprange	2.00E-01	1.08E-01

Table 11: Advection: hyper parameters study

Naive vs Tuned agent benchmarking

Agent	best reward	timesteps	σ	$\bar{r}_t$ over 100 episodes
Naive	-14.13	16M	0.259	- 16.8
Optimised	- 7.409	455k	0.287	-8.5

Table 12: Advection: DRL performances benchmark

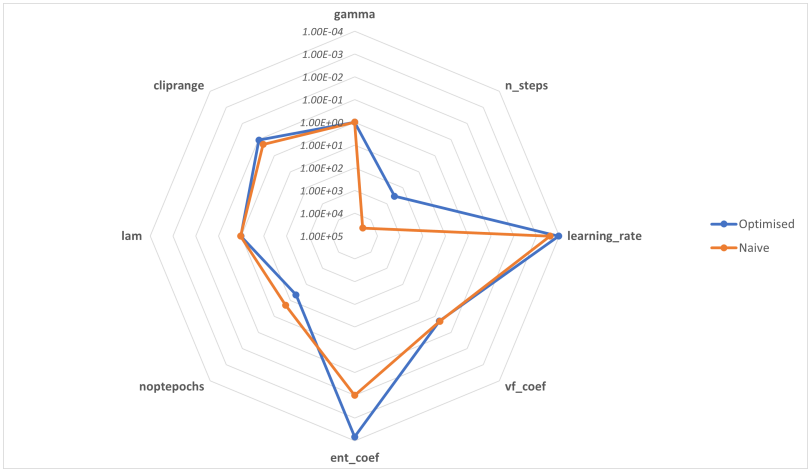

Figure 35: Hyper parameters comparison

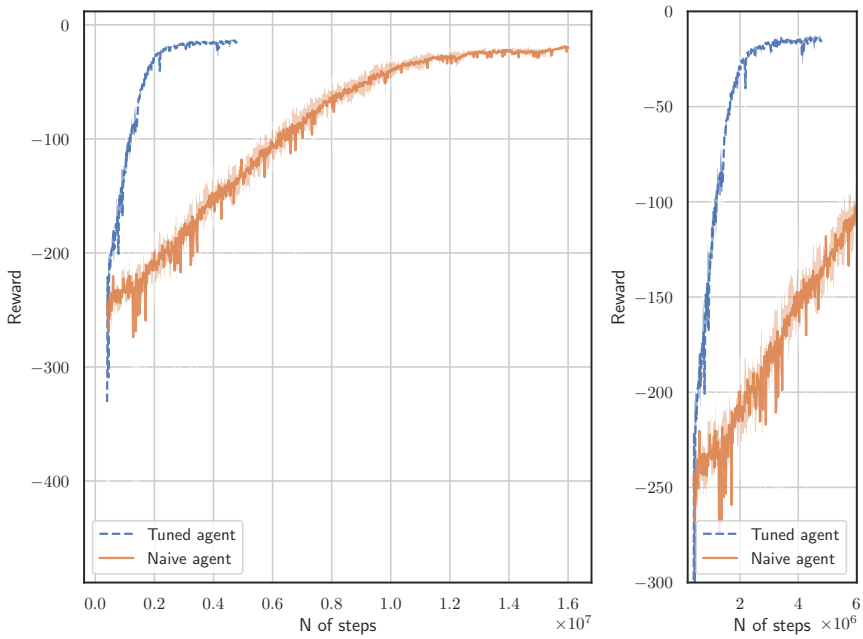

Figure 36: Naive vs Optimised agent

It is recalled that σ is described in eq. (83).

Tab. 12 confirms the influence of the hyper parameters over the DRL agent. Over one hundred episodes, the optimised agent performs better even relying on a reduced number of steps in its training phase. Truth to be told, this number of steps may be misleading since other 85 trials - with the corresponding number of steps - were needed to obtain such a performance.

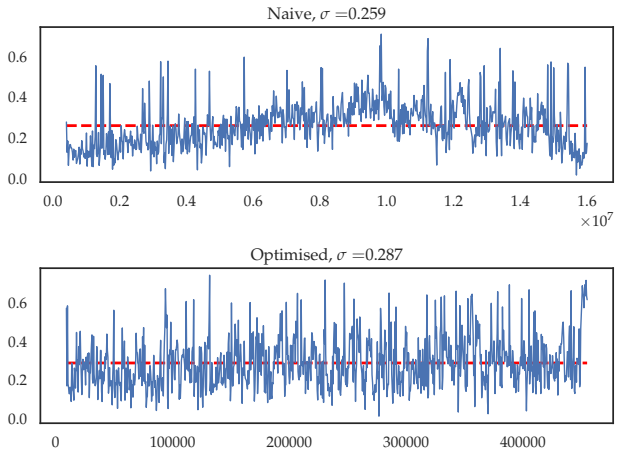

Figure 37: Advection: HPO, detail on dispersions for different initial seeds.

4.1.4 *Optimising the control action*

As a further step of this study, a simpler parametrisation for the control action is employed. In Sect. 2.9 Nelder-Mead and Bayesian Optimisation algorithms were introduced. Hereafter this techniques are used in order to find a proper set of coefficients $\mathcal{C}$ that combined with the observation passed step by step $\mathcal{O}_i$ give rise to the i-th action $\mathcal{A}_i$, where i is the timestep. Hence, the framework is the following:

$$\mathcal{O} \cdot \mathcal{C} = \mathcal{A} \tag{84}$$

It can be preliminary noted how such a *linear* approach is significantly different from the Reinforcement Learning task. The former imposes that the action *has* to be a linear combination between the observed state and some control coefficient to be found, whereas the latter is *discovering* a control law optimising the weights of an Artificial Neural Network that can - theoretically - result in a non linear mapping between the action (output of the net) and the input (observed state).

That being said, this test is conducted in order to assess the differences in terms of performances of the control law and in terms of learning costs using DRL or a much simpler approach.

4.1.4.1 *Nelder-Mead*

Fig. 38 shows the cost history during the optimisation phase. It can be appreciated how almost a monotonic trend is observed. It took *1298* iterations in order to find a set of coefficients that combined with the observations points are able to control the wave. Each iteration is a whole episode inside the Advection Environment as it has been formulated above. In order to give some metric that allows to compare

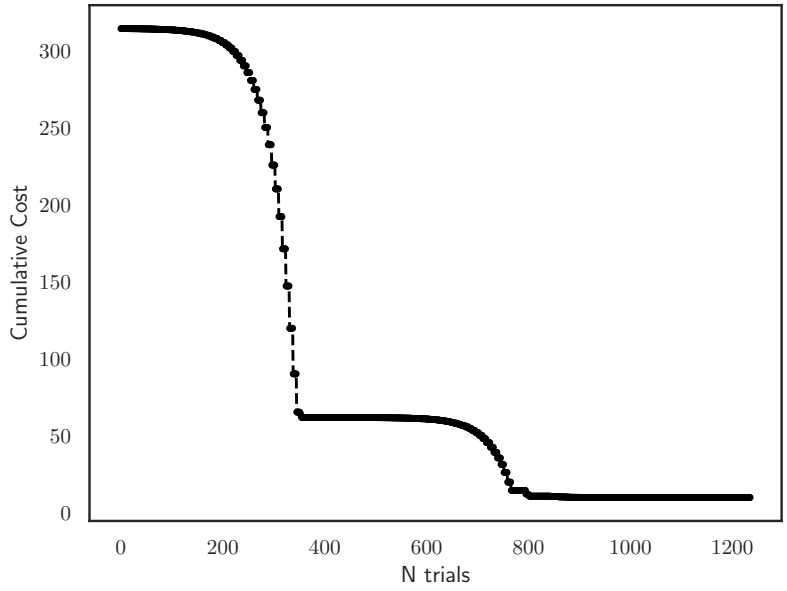

Figure 38: Nelder-Mead Learning Curve

this performance to the DRL one, it can be noted how the total actions (and rewards) to be sampled using this optimization algorithm is:

$$\Sigma_{i=1}^{N_{trials}} \mathcal{A}_i = N_{trials} \cdot \mathcal{A}_{trials} = N_{trials} \cdot \frac{T}{dt}$$

With that the number of episodes being:

$$n_{episodes} = \frac{timesteps}{n.\ of\ steps}$$

Since a trial is defined as a whole wave period and an action is sampled at each timestep. As a result, the Nelder-Mead approach required 514k actions evaluations, achieving a total reward of $r_t = -9.926$ over a episode with this set of coefficients. Finally, it is noted that without the computational burden of training a neural net and being completely gradient free, the simulation lasted roughly 4 minutes on a standard desktop.

4.1.4.2 *Bayesian Optimisation*

Following the same procedure and employing Bayesian Optimisation as a controller, it requires 1168 iterations - i.e.: 431k evaluations - to achieve a maximum reward across an episode of $r_t = -31.27$. Moreover, even increasing the number of maximum iterations the result is not improved, if not slightly.

4.1.5 *Summary*

As a conclusion of this chapter, the two best performances of DRL control are compared with the two obtained with optimisation techniques.

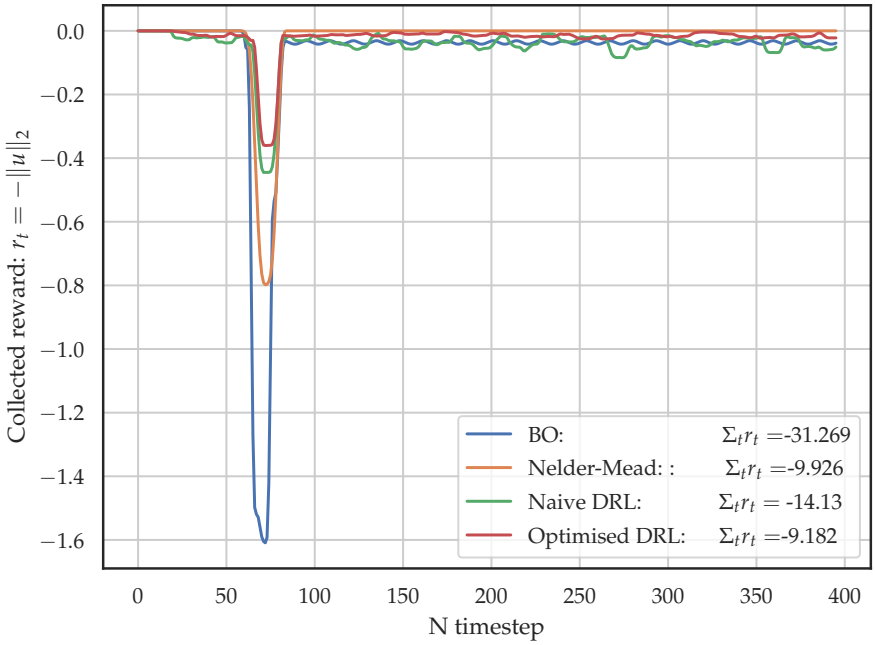

Figure 39: Advection: Rewards collected during an episode, benchmark

Control Performances

Method	$\bar{r}_t$ [a]	total sampled actions	total number of episodes
DRL Naive[b]	−14.13	16M	40404
DRL Optimised	−7.409	455k + x	1149 + x
Nelder-Mead	−9.926	514k	1298
BO	−31.27	431k	1168

Table 13: Advection: control performances benchmark

a Collected reward over 100 episodes
b Here is used the agent that automatically detected when the standard deviation of the collected rewards were under the 15%

4.2 BURGERS EQUATION

The physical set-up of the experiment is the following:

Simulation settings

	description	value
L[m]	length of the physical domain	20
T[s]	simulation duration	5
Nx[-]	space elements	(1e3)
dx[-]	space discretisation step	L/Nx
c[m/s]	wave propagation speed	330
dt[-]	time discretisation step	0.001
Nt[-]	time elements	T/dt
f[Hz]	perturbation's frequency	0.005e2
A[-]	perturbation's amplitude	1e2
C[-]	Courant number	1

Table 14: Burgers simulation setting

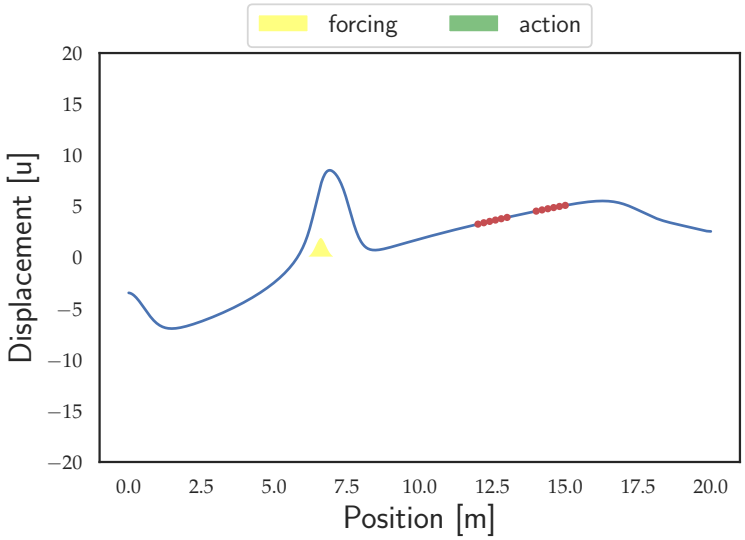

Figure 40: Burgers Equation environment. Overview without any control.

The *Deep Reinforcement Learning* set-up is shown in Tab. 15,
where $\mathcal{A}$ and $\mathcal{O}$ represent the action space and the observations space, using `Python` notation, respectively.

Deep Reinforcement Learning settings

Method	seeds	$\mathcal{A}$	$\mathcal{O}$	reward shape r
PPO	4	$(1,)$	$(36,)$	$\exp\left(-\frac{\|u\|_2}{2\sigma^2}\right)-1$

Table 15: DRL agent settings

The solver implemented is a Crank-Nicolson method solved using the optimised algorithm for banded matrices. For more on this topic, please refer to Sect. 3.5.2.2.

4.2.1 *Neural Network Architecture (NNA)*

Following the same methodology of the Advection Equation study (refer to Sect. 4.1.1.1) the neural network influence on the model performances is studied. As already done for the previous study, the activation function and architecture are changed one at time, starting from the baseline choice of $(64, 64)$ with ReLU.

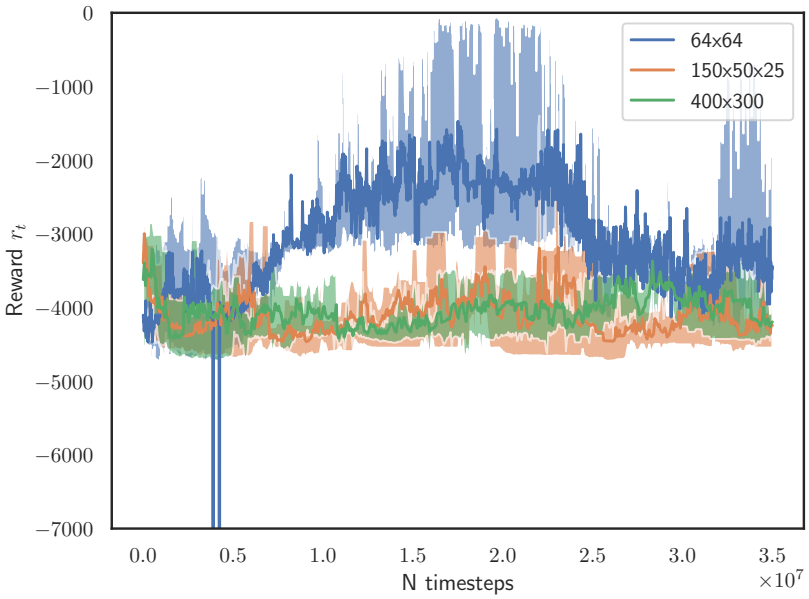

Figure 41: Burgers Environment: influence of the NN architecture

From Fig. 41 it can be seen how the architecture $(64, 64)$ is the one which has the best overall performance. Even though it has a noticeable greater variance in its reward history, it reaches globally highest reward than $(150, 50, 25)$ and $(400, 300)$. Moreover, for the last two architectures, not all four starting seeds lead to an effective control,

while on the other hand $(64, 64)$ does manage to control the environment with the only difference of the time steps required with seeds.

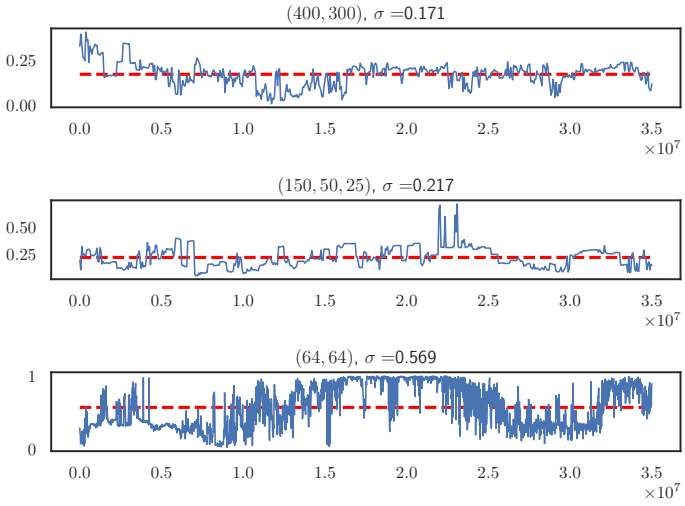

Figure 42: Burgers Environment: dispersion of collected rewards for different NN architecture. Plot of σ per each time step.

Fig. 42 confirms what seen in Fig. 41: the $(64, 64)$ NN has the greater dispersion of the collected rewards over the simulation. This higher variance turns out to be a point of strength for the learning process: being the agent able to reach states that lead it to an high reward, it seeks to come back in such a condition. The other architectures that do not experience such good states tend to remain in this condition, the exploration is not enough.

For what concerns the behaviour of different activation functions, the overall results are reported in Fig. 43. This time the influence is even more important: out of four seeds, the `tanh` activation did not manage to control the wave in all of them. The situation is slightly better for the `leakyReLU` activation, which initial greater exploration lead to the control of the wave in one out of four initial seeds, although not perfectly.

The dispersion across the learning phase, shown in Fig. 44, reflects what said before, adding the information that even if the algorithm embedded exploration, the agent is not able to reach good states that gives good reward and so it is stuck in these bad states.

The behaviour encountered in this section is one of the drawback of the *on-policy* methods, such as Proximal Policy Optimization (PPO), algorithm of choice in this work. The fact that the agent learns directly from its experience it's a problem if it is not able to reach any good state, because it will be stuck in such a region.

As a final note, if the reader finds the trends of leaky ReLU, tanh, $(100, 50, 25)$ and $(400, 300)$ more coarse than the baseline $(64, 64)$ — `ReLU` the guess is correct. In fact, while the number of steps is kept

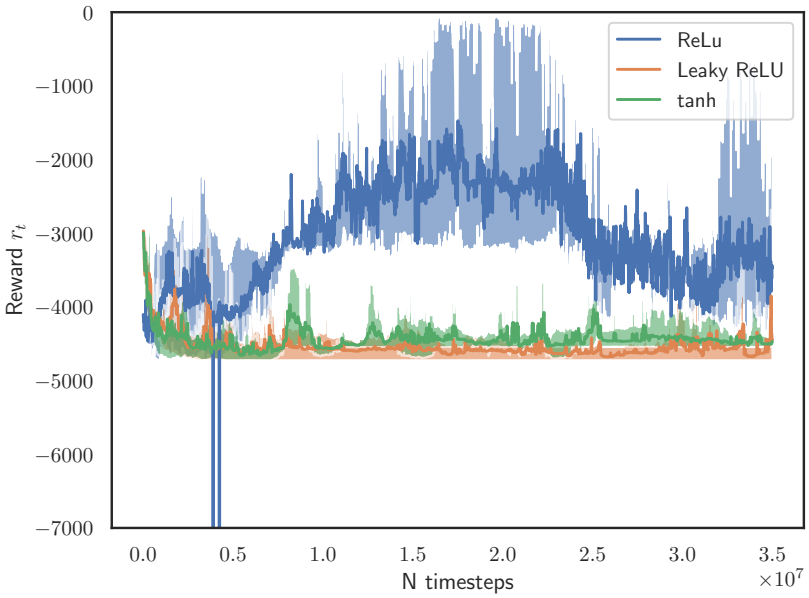

Figure 43: Burgers Environment: influence of the NN activation function

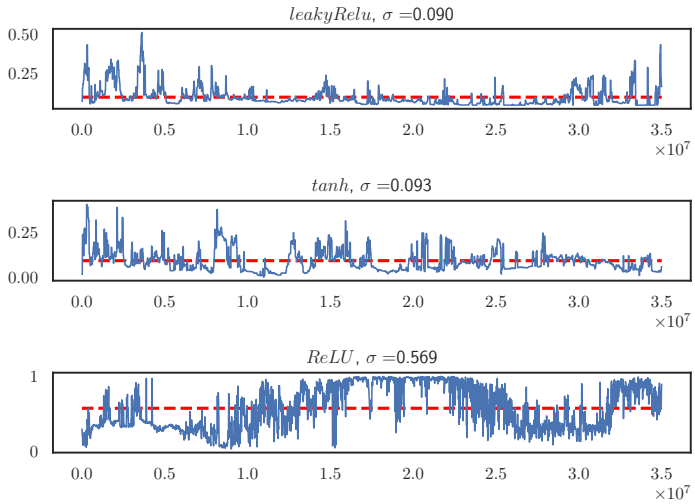

Figure 44: Burgers Environment: dispersion of collected rewards for different NN activations

constant, to compare effectively agents that have been learned for the same number of interaction with the environment, for the simulations of above `Vectorized Environment` have been used, from Stable Baselines ([8]): *Vectorized Environments are a method for stacking multiple independent environments into a single environment. Instead of training an RL agent on 1 environment per step, it allows us to train it on n environments per step.*

4.2.2 *DRL control performances*

The overall setup of the model is reported in Tab. 16.

Main Hyperparameters choices

Method	n. of steps	n. minibatches	noptepochs	learning rate	cliprange
PPO	$3.2 \cdot 10^4$	100	5	$2.5 \cdot 10^{-4}$	0.2

Table 16: Naive PPO simulation parameters

The agent successfully learnt an effective control law in roughly 38M time steps.

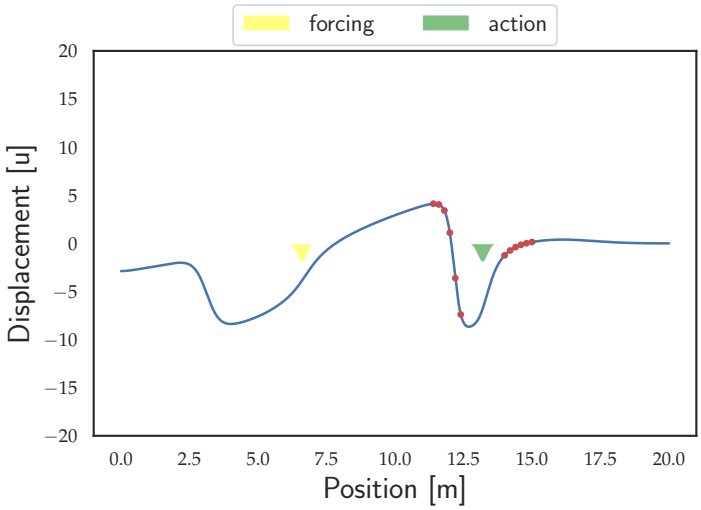

Figure 45: Burgers Equation: controlled wave

Fig. 45 shows this control in action. The external perturbation is almost dumped completely. However, looking at the learning curve (for different initial seeds) in Fig. 46 it can be appreciated how the learning phase is more noisy with respect to the simpler Advection Test case.

In particular, taking aside from the undoubtedly influence on this plot of the reward shape (the reader is referred to Sect. 4.2.5), the learning appears to be more difficult for the agent. In fact, $\simeq$ 24M time steps are needed to learn a good policy. It is also interesting to notice how, after these good results are collected, the quality of the policy gradually deteriorates up to the exit condition. When exiting the simulation - i.e.: the standard deviation of collected rewards across successive episodes are stable under a 15% of tolerance - the agent has learnt a bad policy.

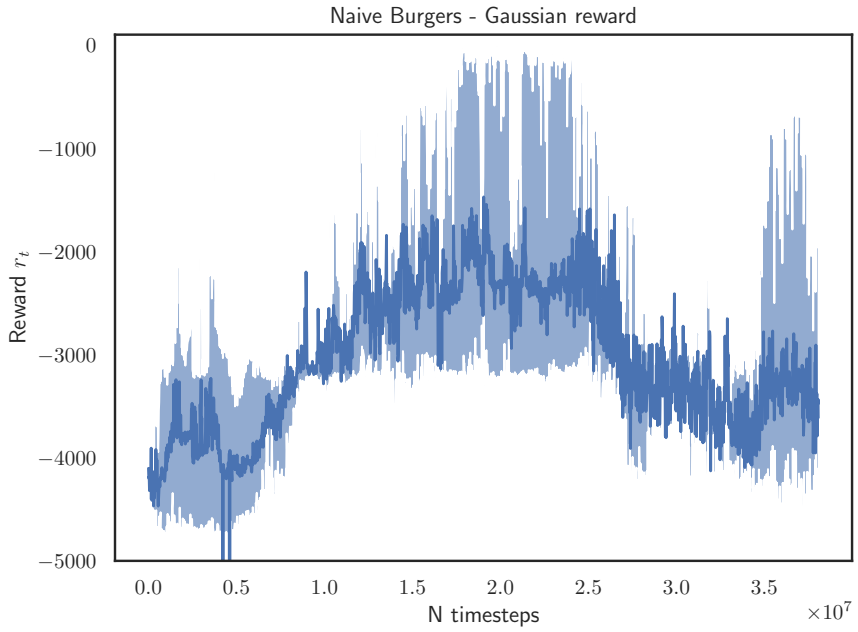

Figure 46: Burgers Equation environment: Learning curve, for four different initial seeds.

Some speculations on that behaviour can be made. Being the RL method belonging to the *on-policy* methods, it is very sensitive to the trajectories in which it happens to end up, as stated in Def. 2.4.3.

Finally, the learning curve in Fig. 46 motivates the approach of using different initial seeds of the Neural Networks at play. It can be appreciated at glance how the performances change deeply with respect to such a condition: for the same time step, a simulation could lead to a completely controlled wave and another is still very far to achieve the control.

4.2.3 *Improving the learning performances: Hyper parameters optimisation*

Following the same steps of Sect. 4.1.3, an optimization of the hyper paramters is carried out, with the goal of improving learning performances.

However, the environment being optimized is slightly different from the one experienced by RL agent. In particular, the time step *dt* has been increased of one order of magnitude to achieve some appreciable results. The optimization performed with $\mathrm{dt} = 0.001$ was unsuccessful, either for the dt being too small or for a insufficient number of trials.

Finally, the hyper parameter space given to the optimizer is the same of illustrated in Tab. 11.

With such a setting, Bayesian Optimization (BO) is able to find a set of hyper parameters good enough to speed up the learning phase. In fact, it took 69 trials to get a set of parameters which, after a learning phase of 1.75M time steps, led to an *overall reward* of $r_t = -83.609$, over a whole episode.

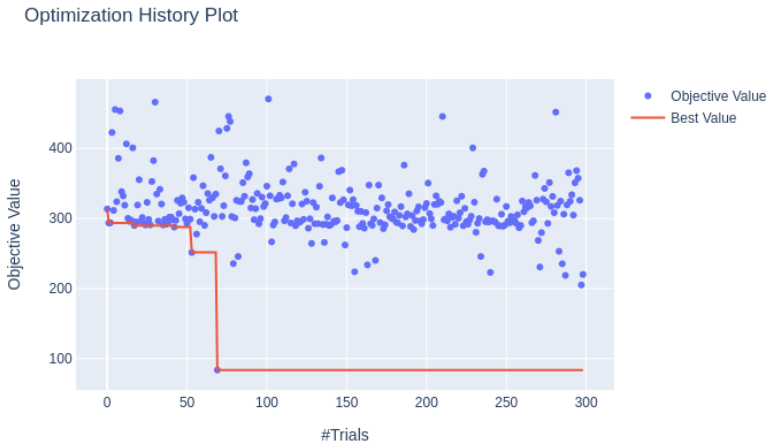

Figure 47: Burgers Equation: optimization history

Hyper parameters comparison

Hyper parameter	Naive	Optimised
gamma*	9.90E-01	9.48E-01
n. of steps	3.20E+04	3.17E+04
learning rate	2.50E-04	5.69E-03
vf. coef	5.00E-01	6.49E-01
ent. coef	1.00E-02	4.69E-04
noptepochs	5.00E+00	3.7E+01
lam*	9.50E-01	9.82E-01
cliprange	2.00E-01	3.23E-01

Table 17: Advection: hyper parameters study

The results shown in Tab. 17 have some similiarities with the best set of hyper parameters obtained for the Advection test case[2] but they differ for almost all parameters. This result highlights *how the best set*

2 These are indicated with the * apex

of hyper parameters is not a unique property of the DRL-agent at hand, but is also greatly dependent on the problem at stake - greatly.

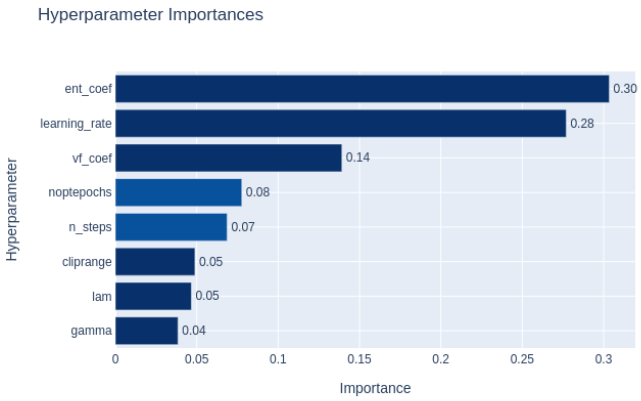

Figure 48: Burgers Equation: relative importance of hyper parameters on learning performances

Notwithstanding what said about the unicity of the set of parameters that yields to the best performance, it is possibile to see some pattern looking at the relative importance of each in Fig. 48 and Fig. 34 - which shows the same plot for the Advection test case. In fact, in both studies *learning rate* and *value function coefficient* have an upmost influence.

This result can be used for setting a further optimization on a reduced search space, hence reducing the compuational burden of the optimizator, focusing on the parameters that influences for the most the learning performances.

4.2.4 *Effects of dimensionless parameters on learning performances*

Exploiting the dimensionless formulation, obtained in Sect. 41,

$$\hat{u}_{,\hat{t}} + \hat{u}\hat{u}_{\hat{x}} = N\hat{u}_{,\hat{x}\hat{x}} + \Phi\hat{f}$$

then the influence of the two dimensionless parameters N, *dimensionless viscosity*, and Φ, *dimensionless inertia*, can be investigated. More generally, since the Burgers equation and its dimensionless formulation have the same formal structure, this study allows to speculate about the impact of viscosity and amplitude (of both control and disturbance) on the learning phase of the agent.

These plots show how the agent generally manages to control the problem, even though there are some parameters set that lead to poor performances.

The overall trend that can be inferred from these figures is that for an increasing intertia parameter (Φ) the agent has some difficulies in

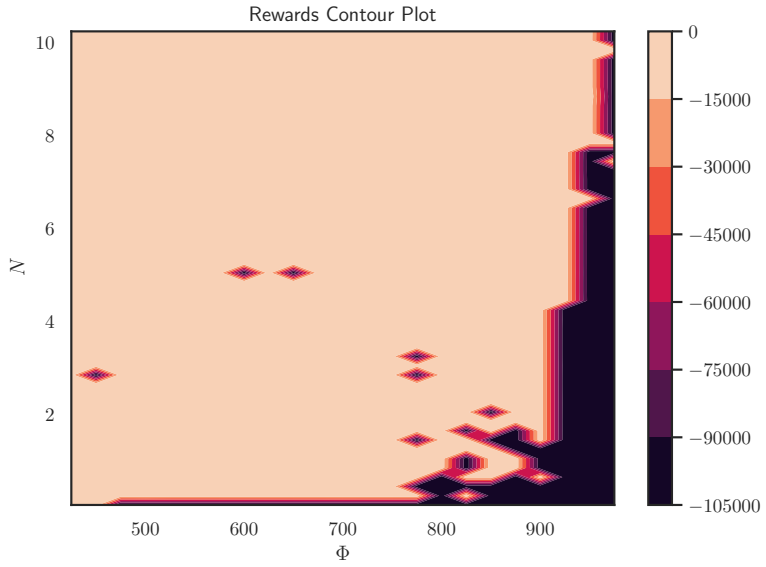

Figure 49: Burgers Equation: dimensionless parameters influence on learning, rewards collected by the trained agent

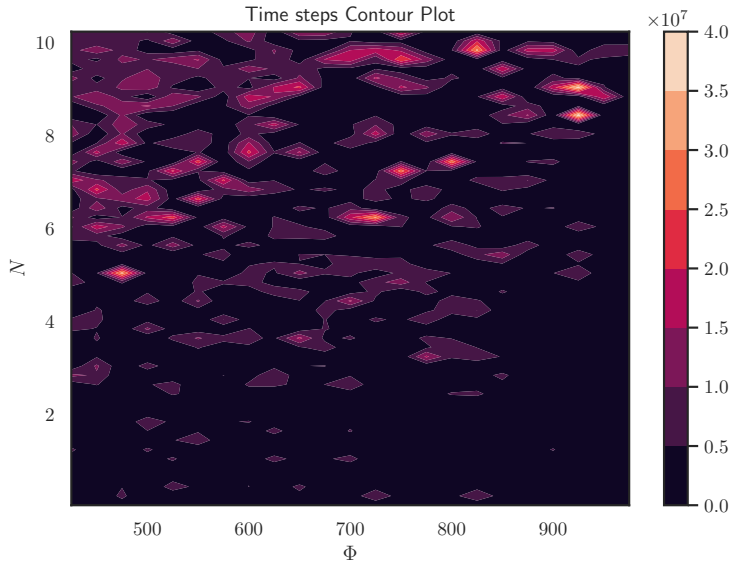

Figure 50: Burgers Equation: dimensionless parameters influence on learning. number of interactions with the environment

converging to a good control-law. At the same time, the dimensionless viscosity (N) dumps these effects, allowing a good control to be discovered.

4.2.5 *A note on the the reward shape influence*

Following the insights provided in [14] the reward shape has been changed from the Euclidean Norm of the displacement, applied in the Advection test case, to the *Gaussian Negative distribution:*

$$r = \exp\left(-\frac{\|u\|_2}{2\sigma^2}\right) - 1 \tag{85}$$

since it has been proved to improve the model performance in the learning phase, applied to the Burgers environment.

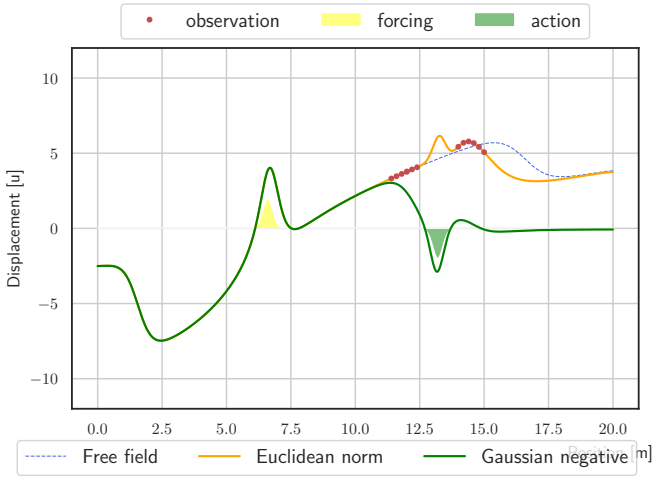

Figure 51: Reward shape influence on model performance. Control applied by two different models implementing *Gaussian negative reward shape (green)* and *Euclidean norm reward shape (orange)*, after 26M time steps of learning.

As shown in Fig. 51, for a fixed time step, the model which tries to maximise the Gaussian negative reward dumps almost completely the external perturbation. On the other hand, if it seeks to maximise the Euclidean norm during learning, its control action is not satisfactory yet.
This behaviour is not seed-dependent.

4.2.6 *Linear Regression of RL Strategies*

Motivated by a search of a deeper understanding of the RL control strategy, the methodology explained in Sect. 2.8 are applied hereafter. In brief, that means that starting from the actions applied for each time step by the DRL agent, $\mathcal{A}_i$, a straightforward pseudo inverse is applied to understand how far the control law effectively learnt by the agent - hence parametrized by a Neural Network - is from a simple linear control. This step can also be interpreted as a linearization of the control law, that is potentially nonlinear, since it has been

parametrised by a Neural Network - a universal function approximator.

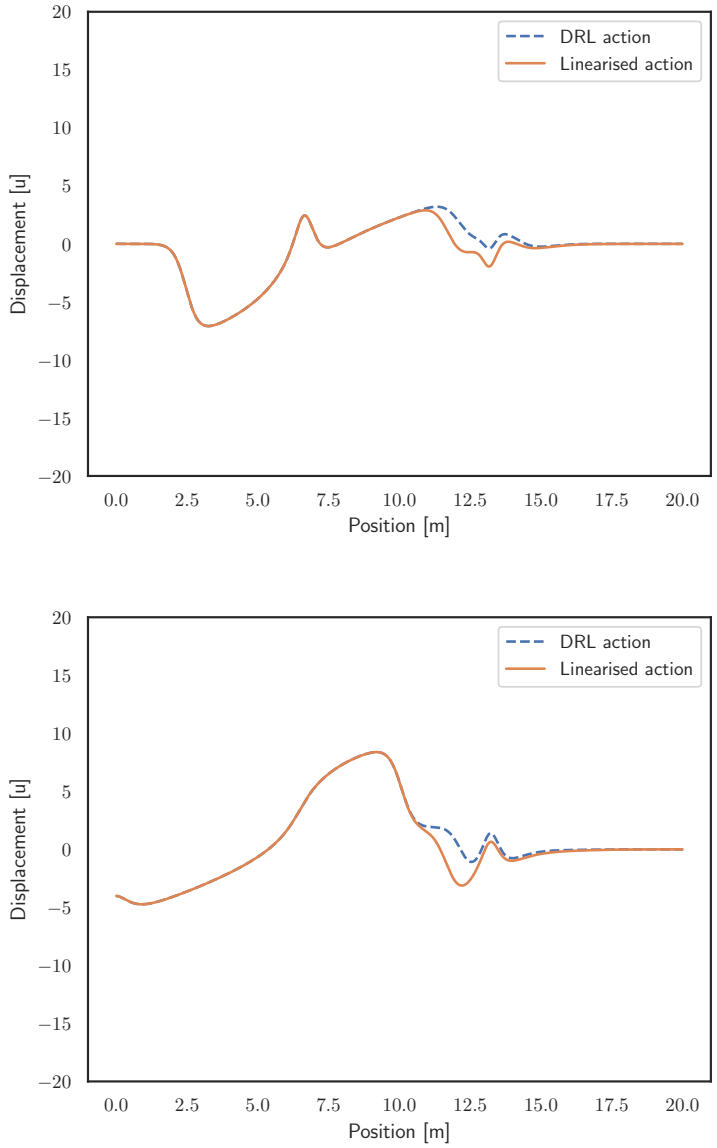

Figure 52: Comparison between linearised-DRL action and DRL one, for different timesteps.

Fig. 52 shows a very unexpected result: the linearised control is very close to the DRL one leading to a residual between the two of `res = 3.38e-01`.
The agent learnt a linear control law, even though the NN parametrization gave to it almost no constraint of the function to be learnt, being NN universal approximators.

It is now of interest to have an estimate of the correlation occurring between the observations and the action took per each time step, to be sure that the ability of NN to generalize is not compromised. To

do so, *Pearson correlation coefficients* are computed for each time step between the observations and the action that from them descends. From an high-level perspective, these coefficients are computed as,

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} \tag{86}$$

where X and Y are the two variables object of study, $cov(\cdot,\cdot)$ represents the covariance between the two and $\sigma_\cdot$ is the standard deviation of $(\cdot)$. A value of $\rho_{X,Y} = 0$ means that there is no correlation between X and Y. Otherwise, if $\rho_{X,Y} = \pm 1$ it would be that X and Y are linearly correlated.

In this computation, X is the $i-th$ observation point, while Y is the action took by the agent at the successive time step - the action that originates from these observations.

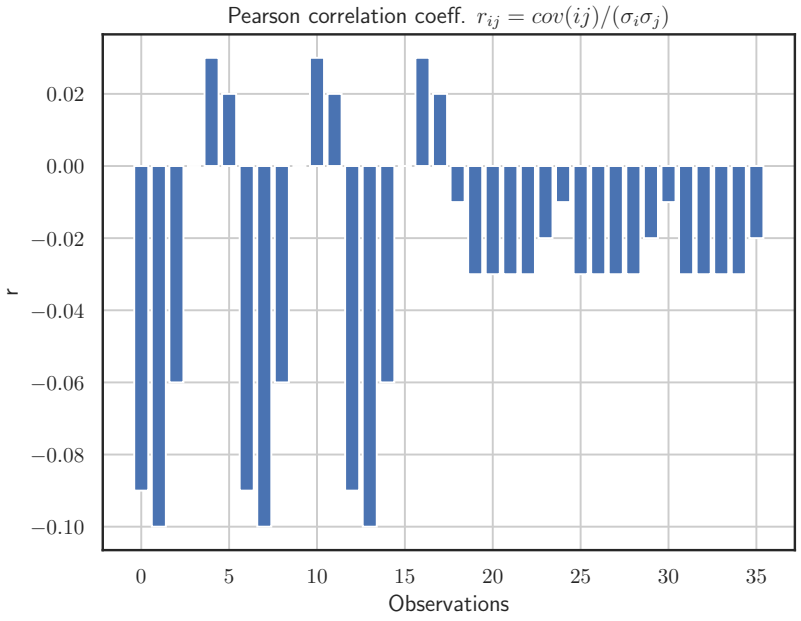

Figure 53: Burgers Equation: Pearson correlation coefficients

As a consequence from Fig. 53 it can be stated that there is a very weak correlation between the observations and actions - if any.

But if the agent learnt a linear control law having the possibility of learn a very non linear one through a Neural Network, a question arise: *What if a simpler parametrization of the control law itself is used?*

This is the aim of the next section.

4.2.7 *Controlling the wave with a simpler parametrization: Bayesian Optimisation*

As a benchmark, the problem of controlling Burgers equation is now attacked with Bayesian Optimisation (BO). In particular, BO is used to

search for the best set of coefficients $\mathcal{C}$ that linearly combined with the observations $\mathcal{O}$ give birth to the action $\mathcal{A}$. The environment is exactly the same of the one used in DRL learning phase, which characteristics are listed in Tab. 14.

It took 1144 episodes - i.e.: 5.72M time steps - for BO to come up with a near optimal set of coefficients $\mathcal{C}$. The control achieved with such a control is almost perfect, collecting a reward of $r_t = -8.603$ *over the whole episode*.

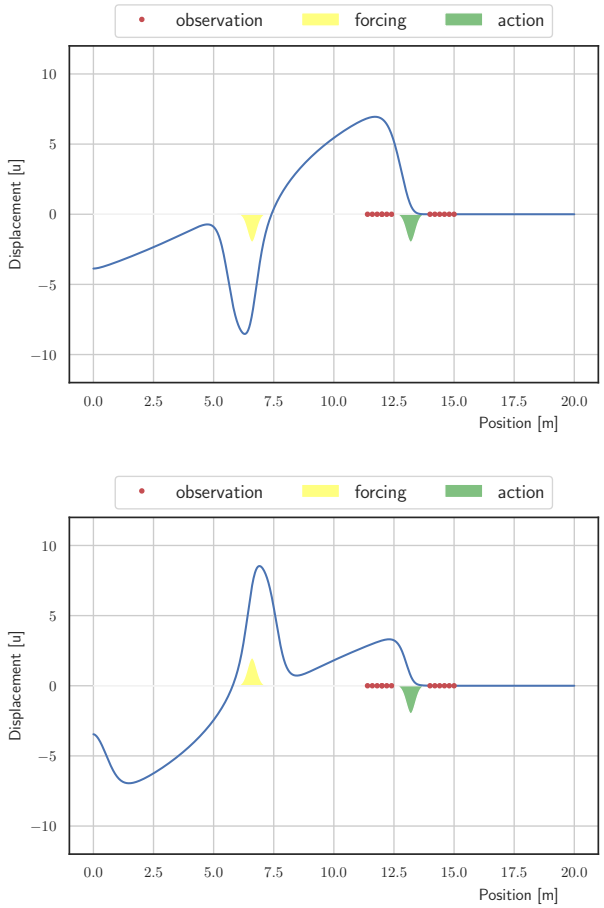

Figure 54: Burgers Equation controlled via Bayesian Optimisation, two different time steps

The two control laws can now be compared. In particular, for each time step, both the BO-control, the DRL-control and the external perturbation magnitude are plotted.

In Fig. 55 a comparison between the two control laws is shown. The overall simulation period is increased from $T = 5[s]$ to $T = 10[s]$ to observe this evolution for more time steps.

The difference between the two methods applied can be detected at glance: while DRL method chooses an action in a stochastic manner for a given distribution that is changing with time, the BO-based

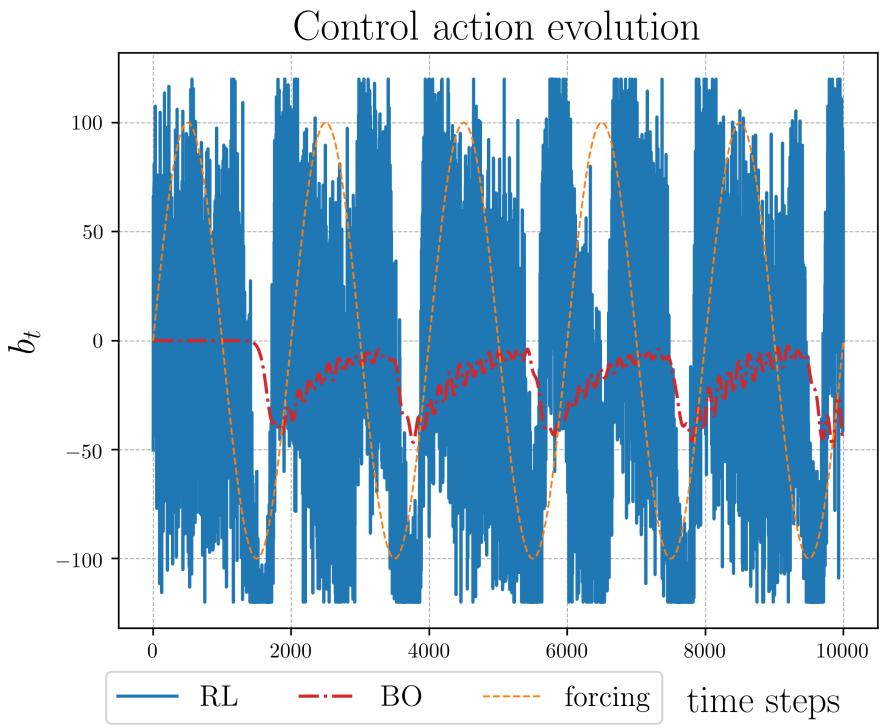

Figure 55: Burgers Equation: comparison between BO-control and DRL-control

approach is fully deterministic since the same coefficients are applied for each time step.

Moreover, looking at Fig. 55, it can be seen how the BO control has a component at the same frequency of the external perturbation - being this control law a linear combination of the observations points, the connection with the frequency of the perturbation acting on the system is preserved - and another one at a more higher frequency. On the other hand, even though the overall trend reminds of a noisy sinusoidal wave, from DRL is difficult to extract any other useful information.

4.2.8 *Summary*

The main results obtained in this test case are summarised in the table below.

Burgers Equation results

Method	time step	$\bar{r}_t$
DRL Naive	$38M^a$	-765.610
DRL Optimised^b	-83.603	$1.75M + x$
BO	$5.72M$	-8.603

Table 18: Burgers simulation results

a The number of time steps needed to effectively end the training phase

b Note that here the time step dt is one order of magnitude greater, increased for optimization reasons. Please refer to Sect. 4.2.3

4.3 CONTROL OF THE VON KARMAN VORTEX STREET AFTER A CYLINDER

In this test case there is no concern in studying the DRL-control performances, that can be easily retrieved in the published work of J. Rabault et al. [24]. Instead, the linearisation of the control strategy is put in to place, in order to assess the quality of a linear control in such an environment.

The *Deep Reinforcement Learning* set-up is shown in Tab. 19,

Deep Reinforcement Learning settings

Method	$\mathcal{A}$	$\mathcal{O}$	reward shape r
PPO	$(2,)$	$(151,)$	$-\lvert < C_D > \rvert - \lvert C_L \rvert$

Table 19: DRL agent settings

where $\mathcal{A}$ and $\mathcal{O}$ represent the action space and the observations space, using `Python` notation, respectively and $< \cdot >$ means the average over the episode.

4.3.1 *Linear Regression of RL Strategies*

Following the same steps of Sect. 2.8, a pseudo inverse is carried out to "linearise" (in a least square sense) the control law of the DRL-agent. In this environment, the agent applies two actions per each time step, these being represented by two jets, Q_1 and Q_2.

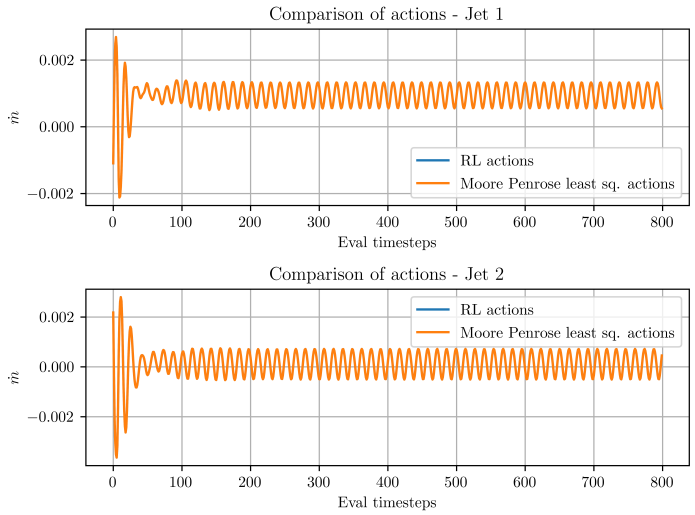

Figure 56: Control of the Von Karman Vortex street after a cylinder: linearisation of the control action

The linearised control superpose *completely* the DRL control law: *the agent is applying a linear control*.

Being the observation space $\mathcal{O}$ so ample, it is now interesting to evaluate this linearisation using a different number of observations point, to understand how the number of observations affects the quality of such a procedure.

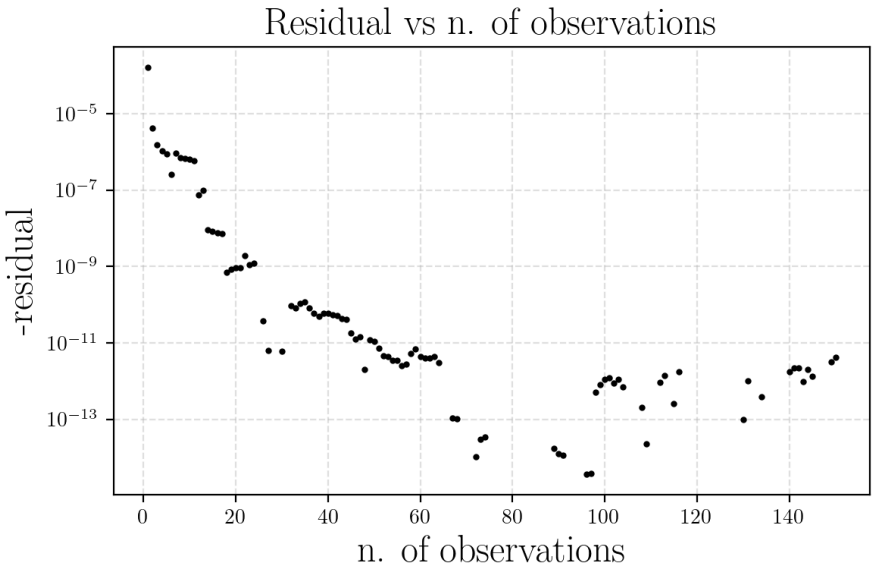

Figure 57: Control of the Von Karman Vortex street after a cylinder: residual between the DRL control law and the linearised one against the number of observations used for such a regression.

As it can be seen from Fig. 57, the residual between the two control laws is extremely low - up to 10^{-5} max -, as already shown in the almost perfect match in Fig. 56. Moreover, this figure shows how, increasing the number of observations points used in the regression, the difference between the DRL-control and a linear one decreases, finding its best spot in the range between 70-100 observations points.

Hence, the same question posed in studying Burgers Equation arise again: *what if instead of the complexity of a Neural Network a simpler parametrisation was employed?*

4.3.2 *Controlling the wave with a simpler parametrisation: LIPO + Nelder-Mead*

As a benchmark, the problem of controlling this environment is now attacked with a combination of Lipschtiz functions and Nelder-Mead. In particular, these optimisators are used to search for the best set of coefficients $\mathcal{C}$ that linearly combined with the observations $\mathcal{O}$ give birth to the action $\mathcal{A}$. The environment is exactly the same of the one used in DRL learning phase, which characteristics are listed in Tab. 19.

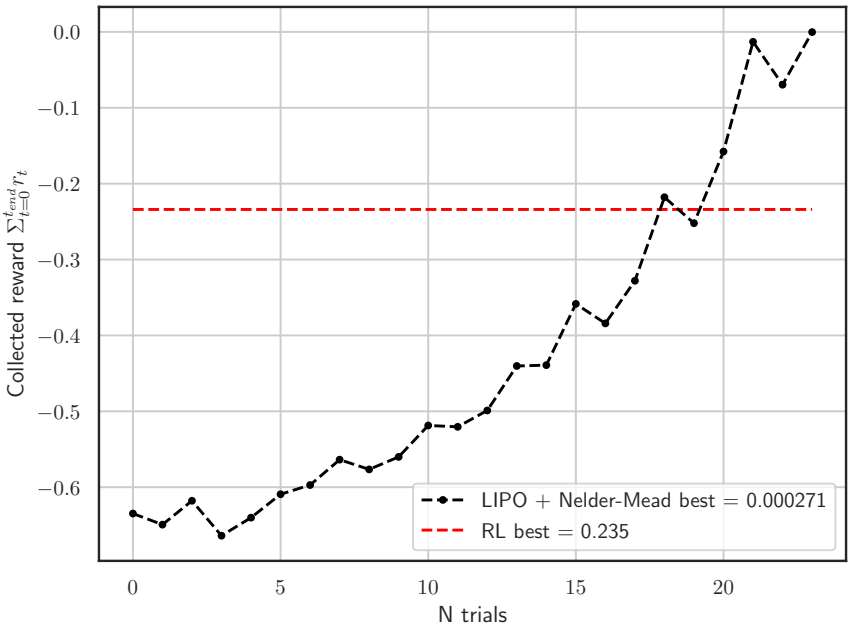

Figure 58: Control of the Von Karman Vortex street after a cylinder: Nelder-Mead trials' cumulative costs

In this simulation, Lipschtiz functions (`LIPO algorithm`) is used until its cumulative reward is not stable. As soon as the trend is stabilising, `LIPO` results are then passed to Nelder-Mead to search for an "optimal" solution in this region.

The optimizer required 39 trials - i.e.: 3120 time steps - in order to achieve such a result.

The instantaneous reward, listed in Tab. 19 is negative by definition. Hence, the agent (or the optimiser, in this case) has the goal of reaching zero as fast as possible. Fig. 58 shows that Nelder-Mead algorithm "learning curve" is almost monotonic in its trend. Moreover, it has to be kept in mind that even if the cumulative reward *over the whole episode* is close to zero, the instaneous reward is not: the optimiser is trying to find a solution that at the end of the simulation gives the cumulative reward closest as possible to zero, as it can be seen clearly in Fig. 59.

Fig. 59 compares the behaviour of the DRL agent and the optimisation results. Looking at the action history with time, the analogy between the two approaches can be seen at glance, especially in the initial, transient phase. After this time, the agent reduces the magnitude of its periodic action. From a *collected reward* perspective the DRL agent is essentially stable close to zero for each timestep - with the exception of the initial transient phase. The strategy applied by the optimiser is somehow different: instead of being close to zero for each time step, it seeks for an *episode reward* equal to zero. As already

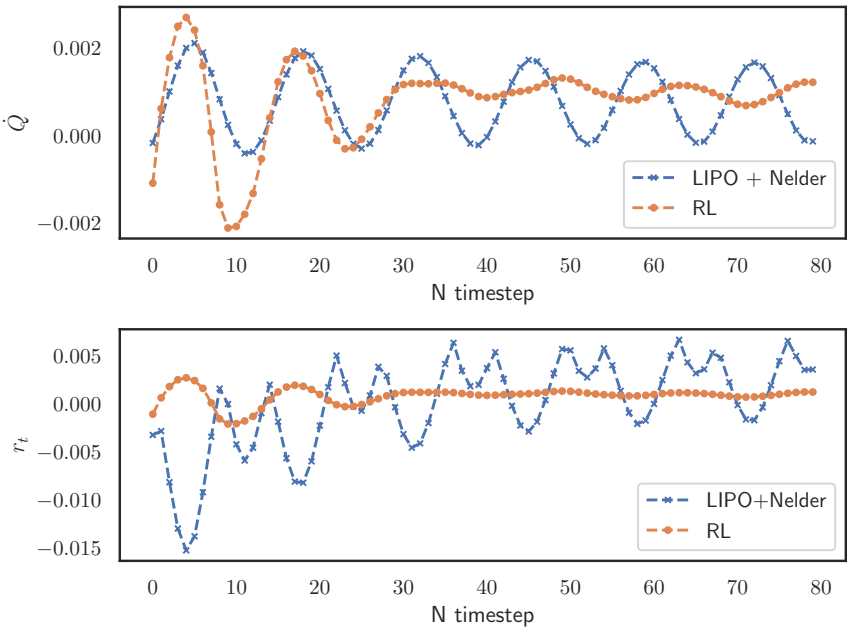

Figure 59: Control of the Von Karman Vortex street after a cylinder: actions and rewards with time

been said before, this translates in a collected reward that changes with time, but which overall sums to zero.

This control strategy is embedded in the time scale of the two experiments. In fact, while the DRL agent receives a reward per each time step, the optimiser collects its reward only at the end of an episode, after n_t time steps.

As a final note, it is underlined how the reward is computed directly from CFD simulations, and summing to this result a proxy coefficient of `0.159` representing the C_D coefficient with no control. For this reason, the reward of both the DRL agent and the optimizer may change sign. This behaviour has not a direct physical meaning, but it is due to how the reward is effectively computed.

4.3.3 *Summary*

Von Karman Vortex street after a cylinder results

Method	time step	$\bar{r}_t$
DRL Naive	16000[a]	0.235
LIPO + Nelder-Mead	3120	0.000

Table 20: Von Karman Vortex street after a cylinder results

a These are the steps required for learning the robust control law which has been compared in this section. However, as reported in the paper of J. Rabault et al. [24], fewer epochs were needed to obtain a less stable control law

5 CONCLUSIONS

The goal of this thesis was to apply Reinforcement Learning in different test cases and evaluate its performances. More interestingly, the control laws learned and applied by the agent in such frameworks were the purpose of this investigation.

The first core result is the highlighted sensitiveness of such a paradigm to a variety of (hyper) parameters. Choices such as the activation functions or the size of the neural networks employed vastly affect the performances of the model; the same is true for the parameters of the model itself.

The second - and unexpected - finding is that, even in nonlinear frameworks such as the Burgers equation or the von Kàrmàn vortex street after a cylinder - environment proposed by J. Rabault et al [24] - the control law put in place by the agent was, with good approximation, a *linear control*. This result raise a question more then answers: was such a complex parametrisation such as neural network needed in the first place? If a linear control law is employed then there are more robust methods for attacking this kind of problems, way more efficient that dealing with the update of neural networks - plural for both Actor and Critic, in this work applications. Statement that is fairly supported by the Results (Sect. 4) of this manuscript. However, when comparing the results of DRL and a simpler parametrisation, the reader has to keep in mind that a fundamental difference occurs between the two: the DRL *learnt* that - in these cases - a linear control law was the most suitable option, while on the other hand when dealing with the optimizers such a control law was forced in the simulation itself.

Finally, it is noted how there is no sake of generalisation in this findings. Large part of these results may be strictly linked with the environment the agent was asked to solve. In fact, it might be that the non linearities the agent faced were not non linear enough to make it chose another, more complex, control law. However that may be, it is clear how DRL may be over-complex for some tasks: while this complexity allow to learn very difficult functions, in simpler environments this complexity turns out to be a burden over the shoulder of the controller.

BIBLIOGRAPHY

[1] Martín Abadi et al. *TensorFlow: A system for large-scale machine learning*. 2016. arXiv: 1605.08695 [cs.DC].

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. *Optuna: A Next-generation Hyperparameter Optimization Framework*. 2019. arXiv: 1907.10902 [cs.LG].

[3] João Carlos Alves Barata and Mahir Saleh Hussein. "The Moore–Penrose Pseudoinverse: A Tutorial Review of the Theory". In: *Brazilian Journal of Physics* 42.1-2 (2011), 146–165. ISSN: 1678-4448. DOI: 10.1007/s13538-011-0052-z. URL: http://dx.doi.org/10.1007/s13538-011-0052-z.

[4] Steven Brunton and Bernd Noack. "Closed-Loop Turbulence Control: Progress and Challenges". In: *Applied Mechanics Reviews* 67 (July 2015). DOI: 10.1115/1.4031175.

[5] J.M. Burgers. *The nonlinear diffusion equation: asymptotic solutions and statistical problems*. 1st ed. D. Reidel Pub. Co, 1974. ISBN: 9789027704948,9027704945.

[6] Facebook. *Facebook Ax Optimisation Toolkit*. https://github.com/facebook/Ax.

[7] Ian Fox, Joyce Lee, Rodica Pop-Busui, and Jenna Wiens. *Deep Reinforcement Learning for Closed-Loop Blood Glucose Control*. 2020. arXiv: 2009.09051 [cs.LG].

[8] Ashley Hill et al. *Stable Baselines*. https://github.com/hill-a/stable-baselines. 2018.

[9] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359 –366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: http://www.sciencedirect.com/science/article/pii/0893608089900208.

[10] Jonathan Hui. *RL - Proximal Policy Optimization*. https://jonathan-hui.medium.com/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12.

[11] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, eds. *Automatic Machine Learning: Methods, Systems, Challenges*. Springer, 2019. URL: https://www.automl.org/book/.

[12] Davis King. *A Global Optimization Algorithm Worth Using*. http://blog.dlib.net/2017/12/a-global-optimization-algorithm-worth.html.

[13] Jens Kober, J. Bagnell, and Jan Peters. "Reinforcement Learning in Robotics: A Survey". In: *The International Journal of Robotics Research* 32 (Sept. 2013), pp. 1238–1274. DOI: 10.1177/0278364913495721.

[14] M.A. Mendez M. Desmet F. Pino. *Reinforcement Learning for Active Flow Control*.

[15] Cédric Malherbe and Nicolas Vayatis. *Global optimization of Lipschitz functions*. 2017. arXiv: 1703.02628 [stat.ML].

[16] Misc. *SciKit, Solve Banded method*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.solve_banded.html.

[17] Misc. *SciPy, Nelder-Mead method for minimization*. https://docs.scipy.org/doc/scipy/reference/optimize.minimize-neldermead.html.

[18] J. A. Nelder and R. Mead. "A Simplex Method for Function Minimization". In: *The Computer Journal* 7.4 (Jan. 1965), pp. 308–313. ISSN: 0010-4620. DOI: 10.1093/comjnl/7.4.308. eprint: https://academic.oup.com/comjnl/article-pdf/7/4/308/1013182/7-4-308.pdf. URL: https://doi.org/10.1093/comjnl/7.4.308.

[19] Fabio Pino. "PhD Intermediate Report". In: (2020).

[20] Python. *the Python tutorial, Classes*. https://docs.python.org/3/tutorial/classes.html.

[21] Python. *the Python tutorial, Functions*. https://docs.python.org/3/tutorial/controlflow.html#defining-functions.

[22] NumPy: the fundamental package for scientific computing with Python. *Moore-Penrose pseudoinverse implementation*. https://github.com/numpy/numpy/blob/v1.19.0/numpy/linalg/linalg.py#L1916-L2012.

[23] Alfio Quarteroni. *Numerical Models for Differential Problems*. Springer International Publishing, 2014. ISBN: 978-88-470-5522-3. DOI: 10.1007/978-88-470-5522-3.

[24] Jean Rabault, Miroslav Kuchta, Atle Jensen, Ulysse Réglade, and Nicolas Cerardi. "Artificial neural networks trained through deep reinforcement learning discover control strategies for active flow control". In: *Journal of Fluid Mechanics* 865 (2019), 281–302. ISSN: 1469-7645. DOI: 10.1017/jfm.2019.62. URL: http://dx.doi.org/10.1017/jfm.2019.62.

[25] Miguel A. Rodriguez, Christoph M. Augustin, and Shawn C. Shadden. "FEniCS mechanics: A package for continuum mechanics simulations". In: *SoftwareX* 9 (2019), pp. 107 –111. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2018.10.005. URL: http://www.sciencedirect.com/science/article/pii/S2352711018300979.

[26] L.S. Andallah Ronobir C. Sarker. "Numerical Solution of Burger's equation via Cole-Hopf transformed diffusion equation". In: *International Journal of Scientific & Engineering Research* (2013), p. 1405. ISSN: 2229-5518.

[27] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. *Trust Region Policy Optimization*. 2017. arXiv: 1502.05477 [cs.LG].

[28] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[29] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. "Taking the Human Out of the Loop: A Review of Bayesian Optimization". In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175. DOI: 10.1109/JPROC.2015.2494218.

[30] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489. ISSN: 14764687. DOI: 10.1038/nature16961. URL: http://dx.doi.org/10.1038/nature16961.

[31] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[32] Aurelian Tactics. *PPO Hyperparameters and Ranges*. https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe.

[33] Yang et al. *Microsoft Neural Network Intelligence*. https://github.com/Microsoft/nni.