POLITECNICO DI TORINO

Master's Degree in Aerospace Engineer
Aerogasdynamics

Master's Degree Thesis

Machine Learning Control on Liquid Film

Supervisors

Prof.ssa Sandra PIERACCINI

Prof. Miguel Alfonso MENDEZ

Candidate

Riccardo GIACOPINO

July 2022

Summary

The jet wiping process is a coating technique that uses impinging gas jets to control the thickness of a liquid layer dragged along a moving strip. This process is fundamental in hot-dip galvanization but it is characterized by an unstable interaction between the gas jet from a nozzle and the liquid film on the strip that results in wavy final coating films. The not homogeneous film thickness is complicate to predict and manage through methods based on fluid dynamics simulations so that here the waves control on the liquid film is treated whereby an innovative concept in engineering: the Reinforcement Learning (RL). Further wiping jets will be controlled by specified algorithms in order to make uniform the liquid film surface. The main advantage of RL is the capability to access a wide range of techniques to define how an agent could decide what action performs to achieve the goal. This work will propose some alternatives: Artificial Neural Network (ANN), Genetic Programming (GP) and function Optimization methods with the purpose to define a virtual agent who chooses an action for the real jet wiping controllers. In RL each algorithm is dependent on hyperparameters so that the reader will understand how much large could be this field of work; the best results will be figured out but additional ways to satisfy the intent of this work could be proposed.

Acknowledgements

I would like to express my gratitude to my primary supervisor, Prof. ssa Pieraccini, who gave me useful recommendations and the honor to write this document in collaboration with the VKI team. I would also like to show my deep appreciation to my advisor, Dr. Fabio Pino, who assisted and guided me throughout this project together with prof. Mendez.
In the end, but not least, I would like to thank my family who without this would have not been possible, and my friends who supported me in the studies.

Table of Contents

List of Tables

List of Figures

Chapter 1

The physical framework

The physical problem we want to manage is related to the control of "undulation" appearing during the hot-dip zinc galvanization process. In the industrial steel manufactory chain, zinc is used to protect the metal from corrosion. Usually, the laminate, moved by wheels, passes through a hot-dip zinc bath and when it goes out no imperfections on the face layer are expected. Once the metal strip is extracted from the bath, an impinging gas jet allows controlling the final coating thickness in a process called *jet wiping*. Although being a very efficient and profitable process, in certain operating conditions the jet starts to oscillate causing an instability called undulation which results in detrimental wavy patterns in the final coating. To cope with this problem, a possible solution consists in using a feedback regulator composed of a set of sensors and gas jet actuators managed by a controller.

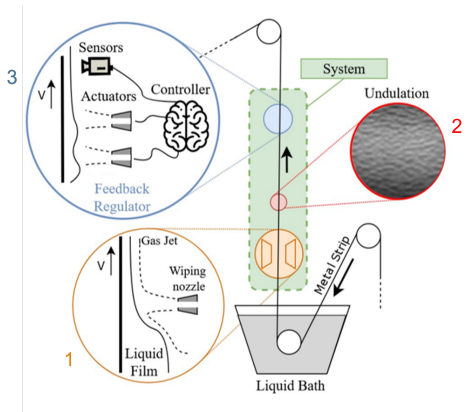


Figure 1.1: *The hot-dip galvanization process and related wiping control: the laminate is immersed in a liquid zinc bath and once taken out, the main jet wipes the amount of zinc in excess (1) generating an undulated distribution of zinc (2) of which we want to control the shape by an active control (3)*

We can use different approaches to set up the controller. The one we are using in this work is based on Machine Learning methods (ML). This pick has been guided by its simple way to work due to the fact that ML is capable to give us a satisfying result by managing just input and output, and no other data or equations are required, therefore its setup cost is reduced. The ML methods need only input, and output at the beginning, so it could be considered a black box that works and acts on the environment by itself. Among these methods, there is a particularly useful branch of ML called Reinforcement Learning (RL).
In a short overview, the basic principle of RL is to create an *Agent* that learns how to achieve a certain goal without human guidance. The agent interacts with the *Environment* outputting an *Action* which goes to modify the environment; the environment produces a *Reward* which carries information about how good has been the action, in terms of how much the new environment state is close to the target.
In RL the agent is blind because it does not know where it is going to but it only tries to maximize what the user specifies as a reward: in fact in RL is effective the *Reward Hypothesis* which assess: *"All goals can be described by the maximization of expected cumulative reward"*.
If we go more in detail, the Agent acquires the Environment State (s_i) by observing it while the last one produces a reward (r) as input for the Learner (Learning Method); the learner is the real "brain" in the process because it elaborates rewards and finds the weights (w) the agent will use in its computation. Using the Environment states and the weights, the agent creates an action (a) used to modify the Environment (to the state s_{i+1}), so a new reward (r_{i+1}) is found and another cycle begins.

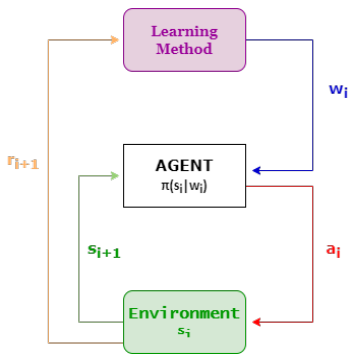


Figure 1.2: Logical flow in Reinforcement Learning

Coming back to the physical problem, the jet-wiping system, some actuators control the zinc film pattern to make it homogeneous. In this work, we will focus on the

jet-wiping actuators.
The jet-wiping actuators will act on the liquid surface after some observations over the thin zinc layer have been acquired: those are the inputs for the Agent.
The observations contain only data about the film thickness and the zinc rate wiped so, step by step, we could relate them to the time evolution of the problem in such a way to figure out the control law which the agent has to perform. This is a reduced-order model (ROM) designed to avoid CFD simulations.
The general kinematic problem could be extrapolated as the following expression:

$$\left\{ \begin{array}{c} \dot{h} \\ \dot{q} \end{array} \right\} = f(h, q) + c(x, t)$$

where $f(h, q)$ is the force perturbing the medium values and $c(x, t)$ represents the control action.
The purpose of the work, as defined above, is to have the minimum thickness for the zinc layer, namely the variable "h", finding the optimal $c(x, t)$ which ensures the minimum value of $\mathcal{L}$ defined as follows:

$$\mathcal{L} = \int_0^t \int_{x_1}^{x_2} h \, dx$$

Minimizing the thickness $\mathcal{L}$ is what ML methods are used for; these allow us to get the best result on their own.
The ML methods include algorithms such as Genetic Programming (GP), Bayesian Optimization (BO) and Lipschitz Optimization (LIPO) that will be taken into account, while the largely used methods are based on Artificial Neural Network (ANN).

Chapter 2

Reinforcement Learning and related algorithms

When we talk about Machine Learning we can think of a large variety of algorithms that can be divided into three big families: the first two families are called *Supervised* and *Unsupervised Learning* with their own typical features explained below. The supervised learning process aims to construct a model from already known inputs and outputs. The user already knows what the solution is for given inputs and he wants to find a model which fits correctly outputs given inputs in order to predict the correct outputs for new given inputs. The learning process requires an initial set of inputs and outputs data used to train the model; as a general trend, the larger the initial data set, the better will be the model predictions for unknown outputs on the new inputs.
We can assess that supervised learning focuses on solving regression problems.
On the other hand, in unsupervised learning, the user knows the input data but not the related output so that the model the user wants to find out is the one useful to predict the output types. The typical example is for a given image input set where we want to understand what kind of subject is shown on each of them to divide the whole inputs data set into different groups.
In this case, we can assess that unsupervised learning focus on a solve the classification problem.

The third family is called *Reinforcement learning (RL)*: this is similar to unsupervised learning in the sense that outputs are usually not given. The RL works by an Agent which interacts with the Environment through actions and then observes the environment's reaction to evaluate a new reward. Reinforcement learning is like trial-and-error learning where is effective the "Reward Hypothesis" and in which the reward R is a scalar feedback signal that indicates how well the agent is doing at step "t".

At each step "t" the agent executes the action A_t, receives observations O_t and the scalar reward R_t while the environment receives action A_t and releases observations O_{t+1} and the scalar reward R_{t+1}. The sequence of observations, actions, and rewards is defined *History* while the *Observation State*, due to the fact that it represents the information used to determine what happens next, is a function of the history.
Given any element, when its *"future is independent of the past given the present"* it could be defined as "Markov": in this case it can be stated that S at time instant "t" is defined "Markov" if and only if: $P[S_{t+1}|S_t] = P[S_{t+1}|S_1, ..., S_t]$.
In short, once the state is known the history may be thrown away, so the current state completely characterizes the process.
Hence, depending on the information known, it is possible to define:

- *Full observable Environment*: in this case the agent directly observes the environment state $O_t := Se_t = Sa_t$ (Observation state corresponds to the environment state which is equal to the Agent state). Formally, when this occurs we are working through a Markov decision process (**MDP**).
  Markov decision processes describe an environment for reinforcement learning. A Markov Decision Process is a tuple (S;A;P;R; γ) where the probability P and the reward R(γ), for the next state, are dependent on action A at the current state S.

- *Partial observable Environment*: the agent indirectly observes the environment (a card-playing agent only observes public cards) hence some beliefs about the environment state needed to be created. This is named as a partially observable Markov decision process (**POMDP**)

The algorithm whereby an agent determines its actions is called *Policy* (π). The policy takes observations as inputs and outputs the action to adopt. The agent should discover a good policy from its experiences with the environment.
A RL agent could include different elements:

I) Policy: agent's behaviour function.
   - Deterministic policy: $a = \pi(s)$
   - Stochastic policy: $\pi(a|s) = P[A_t = a|S_t = s]$.
     This policy type carries out the most probable action after evaluating a probabilistic field. In this case, we talk about "beliefs".

II) Value function $v(s)$: it represents how good is each state/action; it is a prediction of future reward.

III) Model: the representation of the environment used by the agent.
     It predicts what the environment will do next using prediction for Rewards and States.

How can the agent know which of the hundreds of actions it took were good, and which of them were bad? All it knows is what happens after the last action, but surely this last action is not entirely responsible. This is called the *credit assignment problem*: when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it. To tackle this problem, a common strategy is to evaluate an action based on the sum of all the rewards that come after it evaluating a belief reward and usually applying a discount factor γ at each step. The sum of discounted rewards is called the action's return and it is indicated as "G_t". The return corresponds to the total discounted reward from time-step "t".

$$G_t = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k \cdot R_{t+k+1} \tag{2.1}$$

When γ is close to 0 it leads to a "myopic" evaluation, then future rewards won't count for much compared to immediate rewards; conversely, when it is close to 1 leads to a "far-sighted" evaluation, then rewards far into the future will count almost as much as immediate rewards.
The discount factor is important to avoids infinite returns in cyclic Markov processes and moreover, the immediate reward is preferred in animal and human behaviour.

For any Markov Decision Process exists the *optimal policy* π^* that is the best to all other policies ($\pi^* \geq \pi \ \forall \pi$) and all optimal policies achieve the optimal *value function* $v_{\pi^*}(s) = v^*(s)$. To improve a given policy π, iteratively:

1. Evaluate the policy $\pi: \ v_\pi(s) = E_\pi[R_{t+1} + \gamma R_{t+2} + \cdots | S_t = s]$;

2. Improve the policy with respect to $v_\pi: \ \pi' = greedy(v_\pi)$.

Then, some *Predictions* will be carried out to evaluate the future given a policy and perform the *Control* to optimize the future in order to find the best policy (e.g. Policy Iteration/Value Iteration where we find the best value function to obtain the best policy) in accord to the *Principle of optimality: A policy* $\pi(a|s)$ *achieves the optimal value from state* $s: \ v_\pi(s) = v^*(s)$*, if and only if for any state* s' *reachable from* s*,* π *achieves the optimal value from state* $s': \ v_\pi(s') = v^*(s')$.
Generally, each algorithm works combining two steps to achieve the best policy, and the best rewards; these are:

1. *Exploration* which is used to find more information about the environment

2. *Exploitation* which exploits already known information to maximize reward

Finally, as a last notion, the agent could be classified as:

- Policy Based

- Value Based
- Actor Critic (Policy and Value)
- Model Free (Policy and/or Value)
- Model Based (Policy and/or Value and Model)

2.1 Deep Deterministic Policy Gradient (DDPG)

The mathematician Andrey Markov studied stochastic processes with no memory, the *Markov* processes. The probability to evolve from a state s to a state s' is fixed in the Markov chain, and it depends only on the pair (s, s'), not on past states (no memory).
If we think about RL, the aim for the agent is to maximize the return (cumulative reward), which could be defined through the value function, starting from the state s and following the policy π.
Richard Bellman resembled Markov chains but with a twist: at each step, an agent can choose one of several possible actions, and the transition probabilities "P" depends on the chosen action. Moreover, every state transition returns a (positive or negative) reward while the agent is finding a policy that will maximize reward over time. Bellman found a way to estimate the *optimal state value* of any state s, noted as $v^*(s)$, which is the sum of all discounted future rewards the agent can expect on average after it reaches a state s, assuming it acts optimally.
He showed that if the agent acts optimally, then the *Bellman Optimality Equation* applies:

$$v^*(s) = \max_a \sum_s P(s, a, s')[R(s, a, s') + \gamma \cdot v^*(s')] \quad for\ all\ s \tag{2.2}$$

where, given that the agent chooses the action a, $P(s, a, s')$ is the transition probability from state s to state s', $R(s, a, s')$ is the reward that the agent gets in the transition and γ is the discount factor.

This equation leads to an algorithm that can precisely estimate the optimal state value of every possible state: the first step is to initialize all the state value estimates to zero, and then iteratively update them using the *Value Iteration algorithm*:

$$v_{k+1}(s) \leftarrow \max_a \sum_{s'} P(s, a, s')\Big[R(s, a, s') + \gamma \cdot v_k(s')\Big] \quad for\ all\ s \tag{2.3}$$

A remarkable result is that these estimates are guaranteed to converge to the optimal state values v^*, corresponding to the optimal policy π^*.
Bellman found a similar algorithm to estimate also the optimal *state-action* values,

generally called *Q-Values* (Quality Values). The optimal Q-Value of the state-action pair (s, a), noted as $Q^*(s, a)$, is the average sum of discounted future rewards the agent can expect after it reaches the state s via action a, but before it sees the outcome of this action, assuming it acts optimally after that action:

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{n=0}^{N} \gamma^n r_n \mid s_n = s, a_n = a \right] = \mathbb{E}_\pi \left[G_n \mid s_n = s, a_n = a \right]. \tag{2.4}$$

As the same as before for the state value v, the first step is used to initialize all the Q-Value estimates to zero, then update them using the *Q-Value Iteration algorithm*:

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} P(s, a, s') \Big[R(s, a, s') + \gamma \cdot \max_{a'} Q_k(s', a') \Big] \quad for\ all\ (s', a) \tag{2.5}$$

Once we have the optimal Q-Values, defining the optimal policy $\pi^*(s)$ is trivial; in fact when the agent is in the state s, it should choose the action with the highest Q-Value for that state: $\pi^*(s) = argmax_a\, Q^*(s, a)$.
Here we introduce the Q-Learning algorithm as an adaptation of the Q-Value Iteration algorithm when the transition probabilities and the rewards are initially unknown. Q-Learning works by watching an agent play (e.g. random player) and gradually improving its estimates of the Q-Values. Once it has accurate Q-Value estimates, then the optimal policy is to choose the action that has the highest Q-Value: the *Q-Learning algorithm*:

$$Q_{k+1}(s, a) \ \leftarrow \ (1 - \alpha) Q_k(s, a) + \alpha (r + \gamma \cdot \max_{a'} \ Q_k(s', a')) \tag{2.6}$$

where α is the learning rate, an arbitrary parameter.
For each state-action pair (s, a), this algorithm keeps track of an average value of the rewards r the agent gets upon leaving the state s with action a, plus the sum of discounted future expected rewards. To estimate this sum, we take the maximum of the Q-Value estimates for the next state s, since we assume that the target policy would act optimally from then on.

The main problem with Q-Learning is that it does not match well to large (or even medium) MDPs with many states and actions, because evaluating the discounted maximum Q in a wide range of actions and states (e.g. when the state-action pair is a continuous, and not discrete, domain) became resources expensive. The solution is to find a function $Q_\theta(s, a)$ that approximates the Q-Value of any state-action pair (s, a) using a manageable and appropriate number of parameters (given by the parameter vector θ). This is called *Approximate Q-Learning* using deep neural networks can work much better, especially for complex problems. A Deep Neural Network (DNN) used to estimate Q-Values is called a Deep Q-Network (DQN),

and using a DQN for *Approximate Q-Learning* is called Deep Q-Learning.

Consider the approximate Q-Value computed by the DQN for a given state-action pair (s, a). Thanks to Bellman, we know that this approximate Q-Value has to be as close as possible to the reward r that we actually observe after playing action a at the state s, plus the discounted value of playing optimally from then on. To estimate the sum of future discounted rewards, is possible to execute the DQN on the next state s' and for all possible actions a'. We get an approximate future Q-Value for each possible action and then we pick the highest (since we assume we will play optimally) and discount it. This gives us an estimate of the sum of future discounted rewards and by summing the reward r and the future discounted value estimate, we get a target Q-Value for the state-action pair (s, a).

$$Q_{target}(s,a) = \; r + \gamma \cdot \max_{a'} Q_\theta(s',a')$$

With this target Q-Value, we can run some training steps using any Gradient Descent algorithm. Obviously, is considered good practice to try to minimize the squared error between the estimated Q-Value and the target Q-Value.

DDPG is an off-policy method that results in some additional advantages:

1. The off-policy approach can reuse any past episodes ("experience replay") for much better sample efficiency;
2. The sample collection follows a behavior policy different from the target policy, given a better exploration effects.

The behavior policy for collecting samples is predefined, so it is already known and labeled as $\beta(a|s)$. The objective function works summing up the rewards over the state distribution defined by the behavior policy:

$$J(\theta) = \sum_{s \in S} d^\beta(s) \sum_{a \in A} Q^\pi(s,a)\pi_\theta(a|s) = \mathbb{E}_{s \sim d^\beta} \sum_{a \in A} Q^\pi(s,a)\pi_\theta(a|s)$$

given the action domain A and where $d^\beta(s)$ is the stationary distribution of the behavior policy β ($d^\beta(s) = \lim_{t\to\infty} P(S_t = s|S_0,\beta)$) and Q^π is the action-value function estimated with regard to the target policy π (not the behavior policy). Given that the observations used for training are sampled by $a \sim \beta(a|s)$, the gradient could be rewrite as:

$$\nabla_\theta J(\theta) = \mathbb{E}_\beta \left[\frac{\pi_\theta(a|s)}{\beta(a|s)} \cdot Q^\pi(s,a) \nabla_\theta ln \pi_\theta(a|s) \right] \tag{2.8}$$

The policy function $\pi(.|s)$ has been modeled so far as a probability distribution over actions A given the current state and thus it is a stochastic policy; when we

refer to the deterministic policy gradient (DPG), instead, the policy embarks on a deterministic decision: $a = \mu(s)$. It may look weird because the output is a single action when we use a deterministic policy and we miss an actions field where calculate the gradient of the action probability. Hence we call a new objective function to optimize for that is reported below:

$$J(\theta) = \int_S \rho^\mu(s) Q(s, \mu_\theta(s))\, ds$$

with:

- $\rho_0(s)$ is the initial distribution over states;
- $\rho^\mu(s \to s', k)$ is the starting from state s, the visitation probability density at state s' after moving k steps by policy .
- $\rho^\mu(s') = \int_{n=0}^{N} \sum_{k=1}^{\infty} \gamma^{k-1} \rho_0(s) \rho^\mu(s \to s', k)\, ds$ is the discounted state distribution

According to the chain rule, we first take the gradient of Q w.r.t. the action a and then the gradient of the deterministic policy function will be computed ("μ" w.r.t. θ):

$$\nabla_\theta J(\theta) = \int_S \rho^\mu(s) \nabla_a Q^\mu(s, a) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}\, ds \tag{2.9}$$

We can consider the deterministic policy as a special case of the stochastic policy, where the probability distribution contains only one non-zero value over one action.

Given that DDPG combines Policy Gradients with Deep Q-Networks, an Actor-Critic agent which contains two neural networks, a policy network and DQN, is adopted.
The DQN is trained by learning from the agent's experiences. The policy network learns differently (and much faster) than in regular Policy Gradient: instead of estimating the value of each action by going through multiple episodes, the current process works on summing the future discounted rewards for each action, and normalizing them, the agent (actor) relies on the action values estimated by the DQN (critic). We could say that the agent learns with the help of a coach (the DQN).
We can conclude that the Deep Deterministic Policy Gradient is a an algorithm we can ascribe to the model-free, off-policy and actor-critic algorithm, and it combines DPG with DQN (Deep Q-Network) which stabilizes the learning of Q-function by experience replay and a frozen target network β. The DQN usually works in discrete space but DDPG extends its concept to continuous space with the actor-critic framework while it learns a deterministic policy.
In the case of a given deterministic policy, if the agent explores on-policy in the beginning, it would probably not try a wide enough variety of actions to find

useful learning signals. To make DDPG policies explore better, we add noise to their actions during the training. A new exploration policy $\mu'(s)$ is constructed by adding noise N: $\mu'(s) = \mu(s) + N$
In addition, DDPG performs soft updates ("conservative policy iteration") on the parameters of both actor and critic, with $\tau \ll 1 : \theta' \leftarrow \tau\theta + (1-\tau)\theta'$
In this way, the target network values are constrained to change slowly, differently than in DQN where the target network stays frozen for a while.

[1]

2.2 Proximal Policy Optimization (PPO)

PPO alternates between sampling data through interaction with the environment and optimizing a "surrogate" objective function via stochastic gradient ascent. Whereas the standard policy gradient methods perform one gradient update per data sample, PPO is characterized by a new objective function that enables multiple epochs of minibatch updates[1].
The proximal policy optimization (PPO) benefits from some features of the trust region policy optimization (TRPO), but it is much simpler to implement (it is more general) and has better sample complexity (empirically) by introducing an algorithm that achieves the data efficiency and the reliability of TRPO while using only first-order optimization. The objective presents a clipped probability ratio, which forms a pessimistic estimate (i.e., lower bound) of the policy's performance. To optimize policies, PPO alternates between sampling data from the policy (exploration) and performing several optimization epochs on the sampled data (exploitation).
The Policy Gradient [2] methods compute an estimator g of the policy gradient and plug it into a stochastic gradient ascent algorithm.

$$g = \mathbb{E}_t[\nabla_\theta(log(\pi_\theta(a_t, s_t)) \cdot A_t)]$$

where π_θ is a stochastic policy that refers to the parameters θ and A_t is an estimator of the advantage function at time step "t"; the advantage function will be worth deepening later. The estimator g is obtained by differentiating the following objective:

$$L^{PG} = \mathbb{E}_t[log(\pi_\theta(a_t, s_t)) \cdot A_t].$$

The expectation $\mathbb{E}_t[...]$ indicates the average over a finite number of samples (batch), when sampling and optimization alternate. Similarly to the *Trust Region Policy*

[1] Read the reference in Appendix A.1 to deepen into batch size, epoch, etc...

[2] See also Appendix A.2

Optimization (TRPO)[3], the "surrogate" objective function is maximized whereas the size of the policy update is constrained to δ.

$$\underset{\theta}{maximize} \quad L_{\theta}^{CPI} = \mathbb{E}_t\left[\frac{\pi_\theta(s_t, a_t)}{\pi_{\theta old}(s_t, a_t)} \cdot A_t\right] \tag{2.10}$$

$$costrained\ to \quad \mathbb{E}_t[KL(\pi_{\theta old}, \pi_{\theta]})] \leq \delta \tag{2.11}$$

where "CPI" refers to "conservative policy iteration" [4].
Instead of a constraint, we could use a penalty, but it is hard to choose a single value for a penalty that performs well when characteristics change over the course of learning. To achieve a first-order algorithm that emulates the monotonic improvement of TRPO, more modifications are required.
We can use a *Clipped Surrogate Objective* which now considers how to modify the objective, to penalize changes in the policy that move the probability ratio $r_t(\theta) = \frac{\pi_\theta(s_t,a_t)}{\pi_{\theta old}(a_t,s_t)}$ away from 1.

$$L^{CLIP} = \mathbb{E}[min(A_t r_t(\theta), clip(r_t(\theta),1-\epsilon,1+\epsilon) \cdot A_t)]$$

The clipping removes the incentive for moving $r_t(\theta)$ outside of the interval $[1-\epsilon, 1+\epsilon]$. The solution ignores the probability ratio change when the objective improves and includes it when the objective is worse.
$L^{CLIP} = L^{CPI}$ to first-order around θ_{old} , however, they become different as θ moves away from θ_{old}. The probability ratio r_t is clipped at "$1-\epsilon$" or "$1+\epsilon$" depending on whether the advantage is positive or negative.
The loss L^{CLIP} is simply constructed and then we perform multiple steps of stochastic gradient ascent on this objective.
In computing advantage-function estimators we can use a bunch of learned state-value function $v(s)$ introduced into a neural network architecture that shares parameters between the policy (actor) and value function (critic). The loss function has to compute both the policy surrogate and a value function error term and, moreover, it is augmented by an entropy bonus to ensure sufficient exploration. Combining these terms, we obtain a new objective, which will be maximized in each iteration:

$$L_t^{CLIP+VF+S}(\theta) = \mathbb{E}_t\left[L_t^{CLIP}(\theta) - c_1(V_\theta(s_t) - V_t^{targ})^2 + c_2 S[\pi_\theta(s_t)]\right] \tag{2.12}$$

where c_1 and c_2 are two hyperparameter constants coefficients, respectively for the squared-error loss and the entropy, while S denotes an entropy bonus.

[3]Reference in Appendix A.2

[4]Reference in Appendix A.2, KL

The style of policy gradient implementation runs the policy in recurrent neural networks for T time steps (where T is much less than the episode length), and uses the collected samples for an update. This style requires an advantage estimator that does not look beyond time step T. The estimator used is:

$$\hat{A}_t = -v(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} v(s_T)$$

where γ is the discount factor and t specifies the current instant index included in $[0, T]$, within a given $length - T$ trajectory.

In each iteration in the proximal policy optimization (PPO) algorithm that uses fixed-length trajectory segments, N user-defined (parallel) actors collect T time steps of data. Then we construct the surrogate loss on these $N \cdot T$ number of data and optimize it with mini-batch for K epochs.

Algorithm: Pseudo-Code for PPO, Actor-Critic Style

```
for iteration = 1, 2, . . . do
    for actor = 1, 2, . . . ,N do
        Run policy_old in environment for T timesteps
        Compute advantage estimates A_1, ..., A_T
    end for
    Optimize surrogate L wrt θ, with K epochs and minibatch size M ≤ N · T
    θ_old ← θ
end for
```

[2]

2.3 Bayesian Optimization (BO)

Bayesian Optimization (BO) focused on solving the problem $max_x f(x)$. The input x is in R^d and typically $d \leq 20$ in most successful applications of Bayesian Optimizations.
The BO becomes useful when the unknown function $f(x)$ is "expensive to evaluate", in the sense that the number of evaluations is limited to a few hundred because each evaluation requires an amount of time.
The typical form for Bayesian optimization algorithms involves two primary components: (I) a method for statistical inference, which is invariably a *Gaussian Process regression* (GPr), for shaping the surrogate objective continuous function $f(x)$; and (II) an *acquisition function* in order to decide where to sample (i.e. Expected Improvement).
In GPr implementation we observe only $f(x)$ and no its first/second derivatives, so

we refer to the GPr as a “derivative-free" method.
Therefore Bayesian Optimization consists of two main components: a Bayesian statistical model used to model, in fact, the objective function, and an acquisition function for deciding where to sample next. Bayesian optimization chooses to sample next at the point that maximizes the acquisition function.

Algorithm: Pseudo-code for Bayesian Optimization

Place a Gaussian process prior on f
Observe f at n_0 points according to an initial space-flling experimental design.
Set $n = n_0$.
while $n \leq N$ **do**
  Update the posterior probability distribution on f using all available data
  Find x_n, the argmax of the acquisition function over x, where the acquisition function is computed using the current posterior distribution
  Observe $y_n = f(x_n)$.
  Increment n
end while
Return a solution: either the point evaluated with the largest $f(x)$, or the point with the largest posterior mean.

The statistical model, which is Gaussian Process, provides a Bayesian posterior probability distribution that describes potential values for $f(x)$ at a candidate point x where $f(x)$ that is normally distributed with mean $\mu_n(x)$ and variance $\sigma_n^2(x)$.
Gaussian Process regression is a Bayesian statistical approach for modeling functions. We first collect the function's values at a finite collection of points into a vector, from some prior probability distribution; then we evaluate a mean function μ_0 at each x_i, and the covariance function, or kernel (Σ_0), at each pair of points x_i, x_j in order to encode the belief that two closer points should have more similar function values than points that are far apart. Hence, the kernel is chosen so that closer points x_i, x_j in the input space have a large positive correlation.

We suppose to observe $f(x_{1:n})$, for some n, and we wish to infer the value of $f(x)$ at some new points; we let $k = n+1$ and $x_k = x$, so that the *prior* over $[f(x_{1:n}); f(x)]$ is the distribution on $[f(x_1); \ldots; f(x_k)]$ for k points:

$$f(x_{1:k}) \sim \texttt{Normal}(\mu_0(x_{1:k}), \Sigma_0(x_{1:k}, x_{1:k}))$$

where $x_{1:k} = [x_1, \ldots, x_k]$, $f(x_{1:k}) = [f(x_1), \ldots, f(x_k)]$, $\mu_0(x_{1:k}) = [\mu_0(x_1), \ldots, \mu_0(x_k)]$ and $\Sigma_{0(}x_{1:k}, x_{1:k}) = [\Sigma_0(x_1, x_1), \ldots, \Sigma_0(x_1, x_k); \ldots; \Sigma_0(x_k, x_1), \ldots, \Sigma_0(x_k, x_k)]$.
We may then compute the *posterior probability distribution* of $f(x)$ given these observations using Bayes' rule (let see Chapter2 of [3]):

$$f(x)|f(x_{1:n}) \sim \texttt{Normal}(\mu_n(x), \sigma_n^2(x)) \tag{2.13}$$

$$\mu_n(x) = \frac{\Sigma(x_1, x_{1:n})}{\Sigma(x_{1:n}, x_{1:n})} \cdot (f(x_{1:n}) - \mu_0(x_{1:n})) + \mu_0 \tag{2.14}$$

$$\sigma_n^2(x) = \Sigma_0(x, x) - \frac{\Sigma(x_1, x_{1:n})}{\Sigma(x_{1:n}, x_{1:n}}) \cdot \Sigma_0(x_{1:n}, x) \tag{2.15}$$

The posterior mean $\mu_n(x)$ is a weighted average between the prior $\mu_0(x)$ and an estimate based on the data $f(x_{1:n})$, with a weight that depends on the kernel while the posterior variance $\sigma_n^2(x)$ is equal to the prior covariance $\Sigma_0(x, x)$ less a term that corresponds to the variance removed by observing $f(x_{1:n})$.

Kernels, we already said, have the property that points closer in the input space are strongly correlated, that if $|x - x'| < |x - x''|$ for some norm $\| \bullet \|$, then $\Sigma_0(x, x') > \Sigma_0(x, x'')$.
One of the most popular and simple kernel is the power exponential or *Gaussian* kernel,

$$\Sigma_0(x, x') = \alpha_0 \cdot e^{-\|x-x'\|^2}$$

where $\|x - x'\| = \sum_{i=1}^{d} \alpha_i(x_i - x_i')$ and $\alpha_{0:d}$ are parameters of the kernel. Varying this parameter creates different correlations about how quickly $f(x)$ changes with x. The mean function and kernel contain parameters that we typically call the prior hyperparameters and we indicate them via a vector η.
Another commonly used kernel, the *Màtern* kernel:

$$\Sigma_0(x, x') = \alpha_0 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu}\|x - x'\|\right)^{\nu} K_\nu \left(\sqrt{2\nu}\|x - x'\|\right)$$

where K_ν is the modified Bessel function, and we find the parameter ν in addition to the parameters $\alpha_{0:d}$.
We just cited the Màtern kernel for its wide use in BO due to its capability to customize the kernel, but in this work, we will use every time the Gaussian kernel in order to avoid introducing other hyperparameters.

We just discussed the first of the two main components of the BO algorithm, the Gaussian process regression, and what concerns about the acquisition function take place below.
The most commonly used acquisition function is the *expected improvement (EI)* because it performs well and it is easy to use. The literature proposes other acquisition functions but in this work, only the EI will be used.
We suppose to be in the nth iteration when the point x_n is sampled and $y_n = f(x_n)$ is the observed value. The optimal choice (respect the problem on stack $max_x f(x)$) is the previously evaluated point with the highest observed value. Let

$f_n^* = max_{m \leq n} f(x_m)$ be the value at this point, where n is the number of times we have evaluated f thus far.
Now we suppose to have one additional evaluation to perform so that the evaluation at x is $f(x)$. After this new evaluation, the value of the best point we have observed will either be:

$$f_{n+1}^* = \begin{cases} f(x) & f(x) \geq f_n^* \\ f_n^* & f(x) \leq f_n^* \end{cases} \tag{2.16}$$

The improvement of the value at the best observed point is then $[f(x) - f_n^*]^+$ with $y^+ = max(y,0)$. While we would like to choose x so that this improvement is the largest but $f(x)$ is unknown until the evaluation. To cope with this issue we introduce the "expected" value of this improvement and choose x to maximize it. We define the *expected improvement (EI)* as:

$$EI_n(x) := \mathbb{E}_n \left[f(x) - f_n^* \right]^+$$

where $\mathbb{E}_n[\bullet] = \mathbb{E}[\bullet | x_{1:n}, y_{1:n}]$ indicates the expectation related to the posterior distribution given evaluations of f at $x_1, \ldots, x_n$.
As proposed in [4], the EI could be found out as:

$$EI_n = [\Delta_n(x)]^+ + \sigma_n(x) \varphi \left(\frac{\Delta_n(x)}{\sigma_n(x)} \right)$$

where $\Delta_n(x) := \mu_n(x) - f_n^*$ is the expected difference in quality between the proposed point x and the previous best and $\phi(x) = (2 * \pi)^{-1/2} e^{(} - x^2/2)$.
Finally we have the tools to sample the next point, so that:

$$x_{n+1} = argmax \ EI_n(x)$$

[5]

2.4 Liptzich Optimization (LIPO)

Machine learning algorithm has hyperparameters and it is up to the user to determine their values. If we do not set these parameters to "good" values the algorithm will not work well. We all want some black-box optimization strategy like Bayesian optimization to be useful but if we do not set its hyperparameters to the right values it does not work as well as an expert doing guess and check.
A very simple parameter-free for finding the $x \in R^d$ that maximizes a function, $f(x)$, even if $f(x)$ has many local maxima, is Lipschitz Optimization (LIPO). The key idea is to maintain a piecewise linear upper bound of $f(x)$ and use that to

decide in which x evaluate $f(x)$ at each step of the optimization. When the the points $x_1, x_2, , x_t$ are already evaluated, we can define a simple upper bound on $f(x)$ as follows:

$$U(x) = \underset{i=1\ldots t}{min} \quad (f(x_i) + k \cdot ||x - x_i||_2)$$

LIPO picks points at random, checks if the upper bound for the new point is better than the best point seen so far, and if it is so selects it as the next point to evaluate. The method is better than a random search in several non-trivial situations.
If $f(x)$ is noisy or discontinuous at a certain point, the value k could be infinity at that point and it is not going to work reliably since k will be infinity. Not all hyperparameters are equally important, some hardly matter while small changes in others drastically affect the output of $f(x)$. So it would be better if each hyperparameter got its own k.
For a more general treatment, we can also define:

$$U(x) = \underset{i=1\ldots t}{min} \quad [f(x_i) + \sqrt{(\sigma_i + (x - x_i)^T K(x - x_i))}]$$

The noise term σ_i should be 0 most of the time unless x_i is really close to a discontinuity. Here, K is a diagonal matrix that contains the hyperparameter Lipschitz k terms.
The issue is that we do not know the value of the Lipschitz constant k! This is not a big deal since it is easily estimated, for instance, by setting k to the largest observed slope of $f(x)$ before each iteration. We can find the parameters of $U(x)$ by solving an optimization problem:

$$\underset{K,\sigma}{min} \quad \|K\|_F^2 + 10^6 \cdot \sum_{i=1}^{t} \sigma_i^2 \tag{2.17}$$

$$s.t. \quad U(x_i) \geq f(x_i), \qquad \forall i \in [1, \ldots, t] \tag{2.18}$$

$$\sigma_i \geq 0 \qquad \forall i \in [1, \ldots, t] \tag{2.19}$$

$$K_{i,j} \geq 0 \qquad \forall i, j \in [1, \ldots, d] \tag{2.20}$$

where 10^6 is a penalty factor and $\| \bullet \|_F$ is the Frobenius norm.
The σ_i values will be 0 most of the time while still preventing k from becoming infinite, which is the behavior we want.
The final issue that needs to be addressed is LIPO is bad convergence in the area of local maxima. To cope with this issue, we can apply the classic Trust Region method ([6]) to derivative-free optimization.
The Trust Region methods fit a quadratic surface around the best point seen so far and then use the next iterate to maximize that quadratic surface within a certain distance of the current best point. So we "trust" this local quadratic model to be accurate within some small region around the best point, hence the name "trust

region"; It has excellent convergence to the nearest local optima in a very small number of steps.
We can fix LIPO's convergence problem by combining these two methods: LIPO will explore $f(x)$ and quickly find a point on the biggest peak, then the trust region method efficiently finds the real maximizer of that peak. The simplest way to combine these two methods is to alternate between them. On even iterations we pick the next x according to our upper bound while on odd iterations we pick the next x according to the trust region model. Whereas we select the maximum upper bounding point on each iteration we can call this LIPO variation "MaxLIPO"; therefore we can refer to the final algorithm as "MaxLIPO+TR". The upper bound becomes progressively more accurate (approximating the peak), helping to find the best peak while the quadratic model quickly finds a high precision maximizer on whatever peak it currently rests (it finds peaks).
The LIPO's inputs are a number n of function evaluations, a Lipschitz constant $k \geq 0$, the input space X and the unknown function f. At each iteration $t \geq 1$, a random variable X_{t+1} is uniformly sampled over the input space X and the algorithm can decides whether evaluate the function at this point or not. Indeed, the algorithm evaluates the function over X_{t+1} if and only if the value of the upper bound on possible values $UB_{k,t} : x \to min_{i=1...t}\, f(X_i) + k \cdot \|x - X_i\|_2$ computed from the previous evaluations, is at least equal to the value of the best evaluation observed so far $max_{i=1...t} f(X_i)$.
Therefore it is possible explain LIPO as follows.

Algorithm: Pseudo-code for LIPO

Input: $n \in \mathbb{N}^*, k \geq 0, X \subset \mathbb{R}^d, f \in Lip()k$

Initialization: Let $X_1 \sim U(X)$
  Evaluate $f(X_1), t \leftarrow 1$

Iterations: Repeat while $t < n$:
  Let $X_{t+1} \sim U(X)$
  if $min_{i,...t}\,(f(X_i + k \cdot \|X_{t+1} - X_i\|_2) \geq \max_{i,...,t} f(X_i)$ [Decision rule]
    Evaluate $f_{X_{t+1}},\ t \leftarrow t + 1$

Output: Return $X_{\hat{i}_n}$ where $\hat{i}_n \in argmax_{i,...,n}\ f(X_i)$

having introduced some notations and therefore such definitions:

- $X \subset \mathbb{R}^d$ a bounded set;
- $Lip(k) := \{f : X \to \mathbb{R}\ s.t. |f(x) - f(c')| - k \cdot \|x - x'\|_2,\ \forall (x, x') \in X^2\}$ the class of k-Lipschitz functions defined on X;
- $\bigcup_{k \geq 0} Lip(k)$ the set of Lipschitz continuous functions;

- $U(X)$ stands for the uniform distribution over a bounded measurable domain X.

[7] [8]

2.5 Genetic Programming (GP)

Evolutionary Programming (EP) utilizes the concepts of Darwinian evolution to iteratively generate increasingly appropriate solutions (organisms).
Instead of developing a (potentially) complex set of rules which were derived from human experts, EP evolves a set of solutions that exhibit optimal behavior about an environment and desired payoff function.
In a most general framework, EP may be considered an optimization technique wherein the algorithm iteratively optimize behaviors, parameters, or other constructs. "Learning" is a product of the evolutionary process because successful individuals are retained through stochastic trial and error process.
The basic evolutionary program algorithm utilizes four main components: (i) *initialization*, (ii) *variation*, (iii) *evaluation* (scoring), and (iv) *selection*.
The basic EP algorithms starts with a trial population of solutions which are initialized by random or heuristic. The population size, μ, may range over a broadly distributed set but is in general larger than one. After the creation of a population of initial solutions (initialization), variation provides the means for moving solutions around on the search space, preventing entrapment in local minima: Each of the parent member i is altered through the application of a mutation process and generates λ_i progeny which is replicated with a stochastic error mechanism (mutation). Each of these trial solutions is evaluated concerning the specified fitness function. The evaluation function directly measures the fitness, or equivalently the behavioral error, of each member in the population with regard to the environment. The fitness or behavioral error is assessed for all offspring solutions with the selection process performed by one of several general techniques including: (i) the best μ solutions are retained to become the parents for the next generation (elitist) or (ii) μ of the best solutions are statistically retained (tournament), or (iii) proportional-based selection. Finally, the selection process into the population, probabilistically culls suboptimal solutions.
EP differs philosophically from other evolutionary computational techniques such as genetic algorithms (GAs). According to neo-Darwinism, selection operates only on the phenotypic expressions of a genotype; the underlying coding of the phenotype is only affected indirectly. A sum of optimal parts rarely leads to an optimal overall solution: this is key to this difference. GAs rely on the identification, combination, and survival of "good" building blocks (schemata) iteratively combining to form larger "better" building blocks. In a GA, the coding structure (genotype) is of

primary importance as it contains the set of optimal building blocks discovered through successive iterations: we call this process *recombination*. (See Figure 2.1) The building block hypothesis is an implicit assumption that fitness is a separable function of the parts of the genome. This successively iterated local optimization process is different from EP, which is an entirely global optimization approach. In EP the variation operator allows for simultaneous modification of all variables at the same time; therefore fitness is evaluated directly, and it is the sole basis for the survival of an individual in the population. Thus, a crossover operation designed to recombine building blocks is not utilized in the general forms of EP while it is allowed in Genetic Programming (GP).

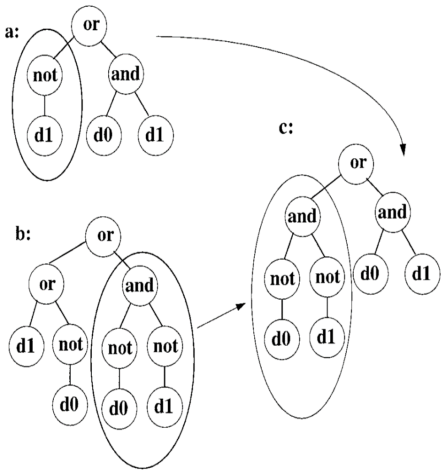


Figure 2.1: *Recombination process*: there are two parent program trees, (a) and (b)recombined through crossover to create an offspring program tree (c). A subtree is chosen in each of the parents, and the offspring is created by inserting the subtree chosen from (b) into the place where the subtree was chosen in (a).

As a general framework for basic instances, we define I to denote an arbitrary space of individuals $a \in I$, and $F : I \to \mathbb{R}$ to denote a real-valued fitness function of individuals. Using μ and λ to denote parent and offspring population sizes, $P(t) = (a1(t), ..., a_\mu(t)) \in I^\mu$ characterizes a population at generation t.

Algorithm: GP pseudo-code:

$t := 0$
initialize $P(0) := \{a'_1(0), a'_2(0), \ldots, a'_\mu(0)\}$
evaluate $F(0) \equiv P(0) : \{\Phi(a'_1(0)), \Phi(a'_2(0)), \ldots, \Phi(a'_\mu(0))\}$
iterate for t
{
recombine: $P'(t) := r_{\Theta_r}(P(t))$
mutate: $P''(t) := m_{\Theta_m}(P'(t))$
evaluate: $F(t) \equiv P''(t) : \{\Phi(a'_1(t)), \Phi(a'_2(t)), \ldots, \Phi(a'_\lambda(t))\}$
select: $P(t+1) := s_{\Theta_s}(P''(t)|F(t))$
$t := t + 1$
}

where:
a' is and individual member in the population;
$\mu \geq 1$ is the size of the parent population;
$\lambda \geq 1$ is the size of the offspring population;
$P(t) := a'_1(t), a'_2(t), \ldots, a'_\mu(t)$ is the populationat time t;
$\Phi : I \rightarrow \Re$ is the fitness mappping;
r_{Θ_r} is the recombination operator with controling paramtetrs Θ_r;
m_{Θ_m} is the mutation operator with controling paramtetrs Θ_m;
s_{Θ_s} is the selection operator $\ni s_{\Theta_s} : (I^\lambda \cup I^{\mu+\lambda})I^\mu$;

Genetic programming is a form of evolutionary algorithm which is distinguished by a particular set of choices as to representation, genetic operator design, and as we already said in fitness evaluation. Fitness evaluation in genetic programming involves executing these evolved programs. Genetic programming, then, involves an evolution-directed search of the space of possible computer programs for ones that, when executed, will produce the best fitness.
To create the initial population a large number of computer programs are generated at random. Each of these is executed and the relative results are used to assign a fitness value to each program. Then a new population of programs, the next generation, is created by directly copying certain, selected by their fitness values, existing programs. This population is filled out by creating many new offspring programs through genetic operations on existing parent programs which are selected based, again, on their fitness. Then, this new population of programs is again evaluated and fitness is assigned to each program based on the results of its evaluation.

Although different representations used to evolve programs exist and the most common is the syntax tree.
Figure 2.2 contains two different types of nodes (as do most genetic programming

representations) which are called primitives (or functions) and terminals. Terminals are usually inputs to the program, although they may also be constants. They are the variables that are set to values external to the program itself before the fitness evaluation is performed by executing the program. Primitives take inputs and produce outputs and possibly produce side-effects. The inputs can be either a terminal or the output of another function.
The tree's leaves, in green, are the *terminals* divided into two subtypes, the constants, and the arguments, while the internal nodes, in red, are the *primitives*.
The primitive type here used is called "Loosely typed" because GP sets as free

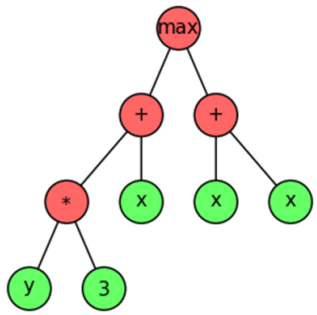


Figure 2.2: GP tree example ([9])

the types between the nodes, so the primitives' arguments can be any element, primitive or terminal, present in the primitive set.
Otherwise, the "strongly typed" primitive set assigns a specified type for every primitive and terminal. The output type of a primitive must match the input type of another one of them to be connected. The generation of trees is done randomly while making sure type constraints are respected.

The DEAP's library contains the main tools to define the tree shape and its operators. The evolution algorithm is defined sequentially.
We first define a "PrimitiveSet()" where we can ".addPrimitive" such as operations whose combination will be evaluated along the evolution. Every operation used in the evolution process is a *primitive* which takes care of its "protection" due to the fact that an invalid element could be used (e.g. when the zero value goes to the denominator in a division and a "ZeroDivisionError" arises).
Once the operators have been defined we just choose the single individual's features. Just to recap, an *individual* is a tree containing a certain amount of primitives (operations) given the inputs (terminals); its features are the tree's depth and how deep each one has to be in each branch. Starting on the single individual we must define how big the whole population is before starting the evolution; the population dimension affects the evolution time and the computing resources required.
In every evolutionary algorithm, for a given population, is mandatory to specify

the method used to *select*, *mate*, *mutate* and *evaluate* an individual into the whole population. (In this specific case the “mate" and “mutate" methods correspond to the *variation* component of the whole algorithm pseudo-code).
Once defined the way to mate, mutate, evaluate and select, these will be described as needed, we have just to use the algorithms provided by DEAP, which are the “simple" algorithm whose pseudo-code is the closest to the one that has been shown before and the $(\mu + \lambda)$ and (μ, λ) algorithms. Specifically, the algorithm works as below:

Algorithm: "Simple" pseudo-code:

inputs: population, the {mate,mutate,evaluate,select} methods
generation=0
evaluate(population)
for generation in $N_{generations}$
    offspring=VarAnd(population, crossover_probability, mutation_probability)
    evaluate(offspring)
    population = offspring
end for

where N_generations means how many times an offspring has to be engendered (or the total number of generations), for a crossover it means the "mate" method defined before, and VarAnd is used to execute the variation part of the algorithm. About VarAnd is worth spending more words on understanding how it works. “VarAnd" contains the variation element in the whole algorithm which performs crossover <u>and</u> mutation. The variation goes as follows. First, the parental population P_p is duplicated and the result is put into the offspring population P_o. The first loop over offspring population P_o is executed to mate pairs of consecutive individuals. According to the crossover probability, the individuals x_i and x_{i+1} are mated. The resulting children y_i and y_{i+1} replace their respective parents in P_o. A second loop over the resulting population P_o is executed to mutate every individual with a given mutation probability. When an individual is mutated it replaces its not mutated version into the population. The crossover_probability and mutation_probability must be in [0,1].
These last two algorithms, $(\mu + \lambda)$ and (μ, λ), are essentially similar to the basic one but in them, the population picked in each iteration is chosen differently.
In $(\mu + \lambda)$ evolution strategy, once the λ offspring from μ parents is engendered, it selects the μ best individuals from the $(\mu + \lambda)$ individuals (parents and offspring) in total.

Similarly as before, here we recommend to observe deeply how the algorithm works and its differences from the simplest one.

Algorithm: $(\mu + \lambda)$ pseudo-code:

```
inputs: population, {mate,mutate,evaluate,select} method, μ, λ
generation=0
evaluate(population)
for generation in N_generations
    offspring=VarOr(population, λ, crossover_probability, mutation_probability)
    evaluate(offspring)
    population = select(population + offspring, mu)
end for
```

One of the main two differences is that the final population is wide as μ and its individuals are chosen from the sum of parents and offspring set. The second noteworthy new concern is the variation method: the name “VarOr" is because an offspring will never result from both operations crossover and mutation. On each of the λ iteration, it selects one of the three operations: crossover, mutation <u>or</u> reproduction. In the case of a crossover, two individuals are selected at random from the parental population P_p, those individuals are mated. Only the first child is appended to the offspring population P_o, the second child is discarded. In the case of a mutation, one individual is selected at random from P_p, it is cloned and then mutated. The resulting mutant is appended to P_o. In the case of reproduction, one individual is selected at random from P_p, cloned, and appended to P_o.
The sum of both probabilities shall be in [0,1] while the reproduction probability is 1 - crossover_probability - mutation_probability.

Differently, the (μ, λ) denotes an evolution strategy that generates λ offspring from μ parents but selects the μ best individuals only from the λ offspring. As a consequence, λ must be necessarily at least as large as μ. However, since the parameter setting $\mu = \lambda$ represents nothing more than a random walk, it is a convention that the abbreviation (μ, λ) evolution strategy always refers to a parametrized strategy according to the relation $1 < \mu < \lambda < \infty$.
In this case, the algorithm is a blend of the previous two and this becomes blatant when observing its following pseudo-code.

Algorithm: (μ, λ) pseudo-code:

```
inputs: population, the {mate,mutate,evaluate,select} methods, μ, λ
generation=0
evaluate(population)
for generation in N_generations
    offspring=VarOr(population, λ, crossover_probability, mutation_probability)
    evaluate(offspring)
    population = select(offspring, μ)
end for
```

The only difference with the $(\mu + \lambda)$ algorithm is the final population returned after each cycle: in this last case the new individuals are chosen only by picking into the offspring set.

[9] [10]

Chapter 3

Applications on GYM's environment

Let's put on scripts what is explained before to understand the representative format of RL
At first, we talked about an environment from which observations are acquired, and then the user-specified agent will manage the reward. We use the gym's tools provided by OpenAI to initiate the process because it provides some already-implemented functions that prove to be very useful in designing policy. Thanks to OpenAI-GYM we can "play" in a classical games environment where the actions space could be discrete or continuous and the performance evaluations should be easy to build.

The tools we were talking before are the same for every environment proposed in which the framework for every algorithm implementation from now on will be featuring by the following steps:

env=gym.make("environment_name") load the environment we want to play. Inside every environment has been defined the output reward.

env.reset() sets the default observations, which in a fully observable environment match with the environment state. Every component of the RL process is now initialized to the default conditions, so the default environment state $S_{t=0}$. The (default) observations state depends on the environment loaded (i.e. in "cart-pole" environment the observation state contains 4 information inside ([poistion, velocity, angle, angle velocity]) while for "mountain car" these are only 2 ([position, velocity]).

env.step(action) for a given action, depending on the environment space into the environment type chosen and, obviously, on the user specified policy, this command evaluate the action effect on the environment state and returns:

- Observations (S_{t+1}): the new observations state caused by the given action;
- The immediate reward r_{t+1};
- Done: it is setted as “True" if the goal is reached or the episode end, else it is “False";
- Info: extra informations.

These facilities' functions stand for the mainstay of algorithm implementation and evaluation. In each problem we will try to manage the environment, once it has been defined, will be reset to the default conditions as soon as the episode begins while along the episode multiple steps will evaluate the action's effect on the environment. In this process, the action changes in accord with the agent computation every time the “.step()" is called and the learning will be visible after a certain episode amount.
One of the best skills that OpenAI-GYM provides is the capability to write a particular one own environment following the basics of the proposed one. This is very useful because in this way it is possible to use the same algorithms used for trivial environments, like the "games" environment, for a more complicated and physical environment as well. This approach will be used in the different environments proposed later.

[11]

3.1 GYM's environments

The models are going to control the two environments released by GYM, Mountain Car and Lunar Lander, whose both action space is continuous and will be defined through a deterministic policy, for which the action is univocally chosen.
Here became necessary to introduce the environment features.

MOUNTAIN CAR CONTINUOUS

Mountain Car Continuous is a “game" environment where the player's purpose is to lead a kart parked in a valley, parametrized like a cosine function, to the mountain peak. The environment state equals the observed space and provides only information such as position and speed whose both domains are finite.
At the beginning the velocity is null and the position is random and obviously before the mountain peak, but when the starting position corresponds to a place far from the valley center, whereas without action, the kart is led by the gravity force to the lower valley point.
Therefore the action domain is continuous and it is included between -1 and 1: the

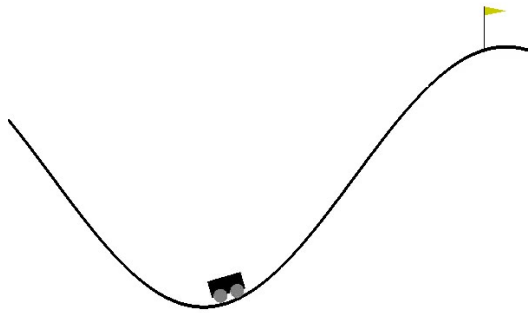


Figure 3.1: Mountain Car environment

action value is a weight for the power the kart use to thrust in the right direction (positive) or the left direction (negative). The reward is negative and proportional to the action, in the way to penalize for taking actions of large magnitude; If the mountain car reaches the goal then a positive reward of +100 is added to the negative reward for that timestep. At each step, position, velocity ad immediate reward are updated:

- $velocity_{t+1} += action \cdot power - 0.0025 \cdot cos(3 \cdot position)$
- $position_{t+1} += velocity_{t+1}$
- $reward_{t+1} += -0.1 \cdot action^2$

The episode end when the kart cross the peak position or the episode complete the maximum steps number (1000).

LUNAR LANDER CONTINUOUS

The Lunar Lander Continuous is another "game" environment where the player's purpose is to land a rocket only controlling what engine fires on and its power. It is a trajectory optimization problem.

In one episode, landing outside of the landing pad is possible and fuel is infinite. The lander at the beginning is located at large above the ground, the top at the center of the screen with a random initial force applied to its center of mass.

Mainly There are four discrete actions available: do nothing, fire the left orientated engine, fire the main engine, and fire the right orientated engine. The action is continuous and included in between -1 and 1, but a direction is figured out to fire on some engines than others.

The environment state is composed of eight information such as: 1) two coordinate data (horizontal and vertical), 2) two-speed data (horizontal and vertical), 3) two angle data (amplitude and angular speed), and other two checks, each for one of

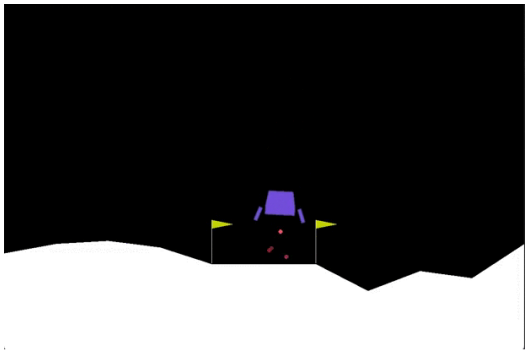


Figure 3.2: Lunar Lander environment

the two rocket's leg, which are "True" (1) when the leg has contact on the ground and "False" otherwise.
When the lander moves away from the landing station, it loses reward points. Firing the main engine is -0.3 points each frame while firing one of the two side engines is -0.03 points each step. Whereas the lander crashes, it receives an additional -100 points else if it comes to rest, it receives an additional +100 points. Additional +10 points are assigned when each leg is on the ground contact. When landing is solved are +200 additional points.
The episode ends when the lander's body gets in contact with the moon, landed rocket, or when the lander crashes, and furthermore when the lander gets outside of the viewport.

3.2 Stable Baselines' tools

Gym provides an environment and functions to choose actions and evaluate the reward, giving us the capability to create a policy. The policy performs actions by learning how to maximize the reward in the lower steps number possible, therefore we care to set the policy to better fit this aim and to make the learning process faster.
Stable Baselines provides effective tools, and an already implemented algorithm, to maximize the total reward to find the best policy. Among the algorithms provided by Stable Baselines, here have been chosen the PPO and DDPG.
One of the first methods used is called PPO1: this algorithm is the first version of the proximal policy optimization method because another method called PPO2, similar but improved with another kind of learning rate value used, is also provided by Stable-Baselines.
These two methods use both the clipping method to avoid too large update and

are called down with the common following inputs:

model = PPO# (

- The "Mlp" (Multi-layer-perceptron) for actor-critic composed by 2 layers of 64 perceptrons,
- The environment name loaded with gym.make(),
- The discount factor γ with default value equal to 0.99,
- time steps per actor per update (128 for #=2, 256 for #=1 as default),
- The clipping parameter ϵ with default value equal to 0.2 **)**
- the entropy loss constant with default value equals to 0.01,
- The optimizer's number of epochs wiith default value equals to 4,
- The optimizer's batch size (64 for #=1 and update_steps/4 for #=2 as default),
- The advantage estimation with default value equals to 0.95,
- "policy_kwargs": additional arguments to be passed to the policy on creation. In this work it is used to vary the artificial neural network dimension when we don not want use "Mlp".

model.predict(observation) : this method return the action and a prediction of the new environment state from the model, given an initial environment state (equivalent to observations due to the deterministic policy),

model.learn(total_timesteps) : this command launch the learning process, so the forward steps and back-propagation to define the weights inside the ANN. Here the total_timesteps are the total number of samples to train on, the number of how many times perform the ".step()" method proposed by the Gym environment.
Given a large number for timesteps, the model chosen will be called "total_timesteps"┼ times. Higher is the input more precise the network predictions will be, but increasing the input implies a more time required in learning process.

Above only the entropy coefficient c_2 has been indicated so it means that in PPO the relative error loss term between policy and value function is not corrected. However this is true only for PPO1 because PPO2 take as input also $vf_{coef} \equiv c_1 = 0.5$. Moreover, While PPO2 takes in input also the "learning_rate" (0.00025 as default), PPO1 uses the ADAM method to evaluate it and the values "optim_stepsize" (the ADAM "alpha") and "adam_epsilon" (respectively equal to 0.001 and 1e-05, in default mode) should be changed as we wish in PPO1.

PPO1 works by updating the actor every 256 steps (actor batch-size) and it processes 4 epochs, whose batch size is 64 steps to conclude the optimization process. PPO2 uses similar values except that they are not absolute but relative to the surrogate objective: the optimization batch size is composed of 128 steps and it uses mini-batches per update (the total value divided 4 times as default) and the same for the epochs which are mini-epochs when optimizing the surrogate (again the total divided 4 times as default)

The DDPG is also used in this work so here its parameters are presented like for PPO#.

model = DDPG (

- The "Mlp" (Multi-layer-perceptron) for actor-critic composed by 2 layers of 64 perceptrons,
- The environment name loaded with gym.make(),
- The discount factor γ with default value equal to 0.99,
- The $\tau = 0.001$ for soft actor and critic update,
- The batch-size with default value equals to 128,
- The interval of steps to apply the noise is 50 as default,
- The actor learning rate with default value equals to 0.0001 as default,
- The critic learning rate with default value equals to 0.001 as default,
- The buffer size with default value equals to 50000 as default,
- "policy_kwargs": additional arguments to be passed to the policy on creation. In this work it is used to vary the artificial neural network dimension when we don not want use "Mlp".

In DDPG the policy learns in accord with the batch-size of 128 steps and buffer size of 50000. The user can provide a sort of Noise, defining it previously and this is going to be used by the algorithm every 50 time-steps to ensure a better exploration .

It is possible to save the model weights after the training with a particular name via "*.save('name')*" and then load in the future the model already trained with "*.load('name', env)*".

3.3 Application of LIPO

Here we try to find the global maximum for the reward distribution, which is the objective function, in this case, using the LIPO algorithm with the Trust Region

method (MaxLIPO+TR).
In this case on stack the useful tools are provided by the "dlib" library (See [12]), and the method used in this implementation are:

global_function_search() implements both the LIPschitz Optimization and Trust Region Method.

function_spec(for_each_weight) define the domain exploration space for each weight. We have to specify the interval set of each weight, which means the maximum and minimum value it could assume.

set_seed() to seed a bunch of points as the first approximation to initialize the optimization.

get_next_x() to pick another point where evaluate the objective function. Here the new "point" is an array containing the weights picked inside their exploration domain defined before.

set() is used to load on the algorithm the function value evaluated at the current "point". In this way, receiving the reward, the algorithm understands how good is the weights set (the "point") picked at the current iteration.

The objective function to maximize is the episode total reward, corresponding to the sum of the single reward returned from the ".step()" function at each step, until the variable "Done" became "True" (this occurs in different ways for different environments).
All Gym's environments produce a new observations array (the new environment state) $S_t = [s_{1,t}, \ldots, s_{n,t}]$ with n elements after each time-step t (the current step t after the method ".step()" has been called t-times), but the new observation is the product of an action evaluated through stepping ahead, so how to define the action??
How we know the policy function is defined by the user to discern what action output after each observation (action and policy correspond due to the deterministic representation wanted), but in this case, the policy is implicit and defined through the weights' array $w_t = [w_{1,t}, \ldots, w_{n,t}]$ at the current time-step t, found out by the algorithm, in linear combination with the observation state.

$$a_t(S_t, w_t) \equiv \pi(S_t, w_t) = w_t \cdot S_t^T$$

For more general purposes, the weights chosen by the algorithm could be used for different computing types, exponential, harmonic, or whatever else non-linear function. To be precise, in this case on stack, some arbitrary weights linearly multiply the observations array's elements to evaluate a new action, so the optimization

consists in finding the weights at the highest total reward iteratively so that new actions are taken on the track of the best score.

We can apply LIPO to the two game environments presented before. in both, we will use the linear combination to define the action.
In the Lunar Lander Gym's environment is important to be careful with the observation array's length, because the dimension of the weights needs to adapt with it. Moreover, the action is not unique because here we have to control three engines to define the speed and direction and no more only one like Mountain Car.

3.4 Application of BO

The second optimization algorithm is the Bayesian Optimization and here an implementation of this algorithm is going to be proposed.
The script framework is very similar to the previous for LIPO but the tool used is the "skopt" library "Optimizer". In this case, the "Optimizer" compute the Gaussian Process (GPr) to evaluate the next point, so the next function value but the user has to define also the hyperparameters: kernel and the acquisition function.
Unlike LIPO, whereas we refer to the BO theory, this algorithm tries to minimize the loss function, so it is a minimizer actually, instead of the maximizer LIPO.

Optimizer mainly takes in inputs:

- Domain space for each weight similar to the LIPO approach,
- the "base_estimator" or "Kernel". In this work, we will use the default Kernel which is the Gaussian Kernel but one can choose the one he wants trying to customize that,
- the acquisition function for deciding where sample next. In the next results the "acq_func" chosen is the EI (expected improvement),
- a user-specified number of initial points where evaluate the objective function for the prior evaluation with a default value equal to 10,
- the generator type, or rather the method to deploy the number of initial points: the default is the random method.

ask() is useful to get a new "point x", so a new set of weights, through which evaluate the objective function;

tell(x, reward) using for recording the objective function value related to the current point, the set of weights.

Similarly to the LIPO optimization, a linear combination is used to combine state and weights in this case.
Step by step the two Optimization algorithms work sampling a new point, a set of weights, to compute a new action in order to evaluate the reward; the reward information leads the sampling process in order to find the global maximum, or minimum, but the learning speed depends on the algorithm on stack.

[13]

3.5 Algorithms comparison

We used the game Gym environment, Mountain Car and Lunar Lander, to run the algorithms widely discussed above.
The optimizators, LIPO and BO, pick the weights' array element from a continuous set space bounded by ±50 so that $w_{i,t} \in [-50{,}50]\ \forall i \in [1, \ldots, dim(w_t)]$.
The objective function is the total reward for the whole episode taken as sum of the immediate reward after each step.

$$J(a_t) = \sum_{t=0}^{T} r_t(a_t|S_t)$$

where t is the current time-step and T represents the total time steps per episode. It is noteworthy to remember LIPO is a maximizer while BO is a minimizer, so for BO the actual $J_{BO}(a_t) = -J(a_t)$.
As for the algorithm which uses ANN (PPO#, DDPG) we use their default parameters except that for DDPG where we specify the noise parameter, the Ornstein-Uhlenbeck noise, which represents a Gaussian noise distribution in short. The performances attributed to Figure 3.3 are shown in Table 3.1.

	Mountain-Car					Lunar-Lander				
	PPO1	PPO2	DDPG	LIPO	BO	PPO1	PPO2	DDPG	LIPO	BO
reward	0	$-0.6 \cdot 10^{-3}$	96.2	99	98	67	252	306	297	2575
episodes	500	500	3691	1500	500	832	117	1149	1500	500
time [min]	7.7	8.4	21.8	2.5	32	14.3	17.3	31.6	7.5	3.71[h]

Table 3.1: Algorithms' performances on GYM-environments

The total number of time-steps set in the stable baselines algorithms learning process are $5 \cdot 10^5$ but the relative episode length could change along the learning

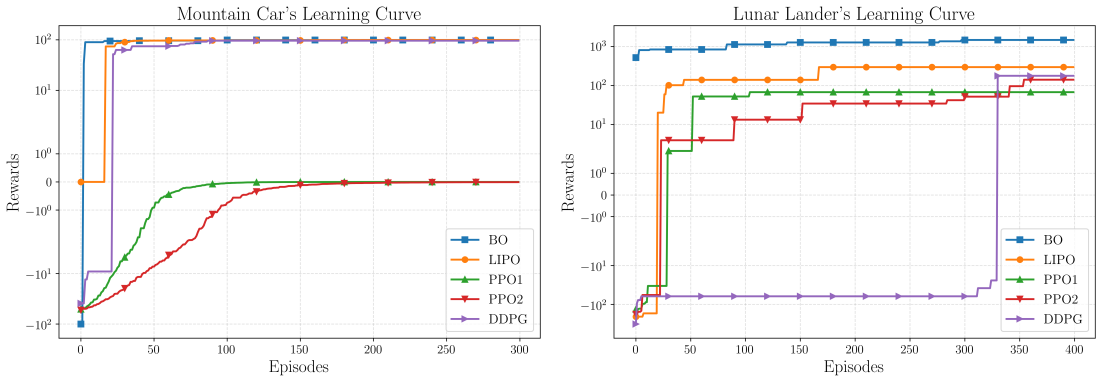


Figure 3.3: Learning Curves for Mountain Car and Lunar Lander

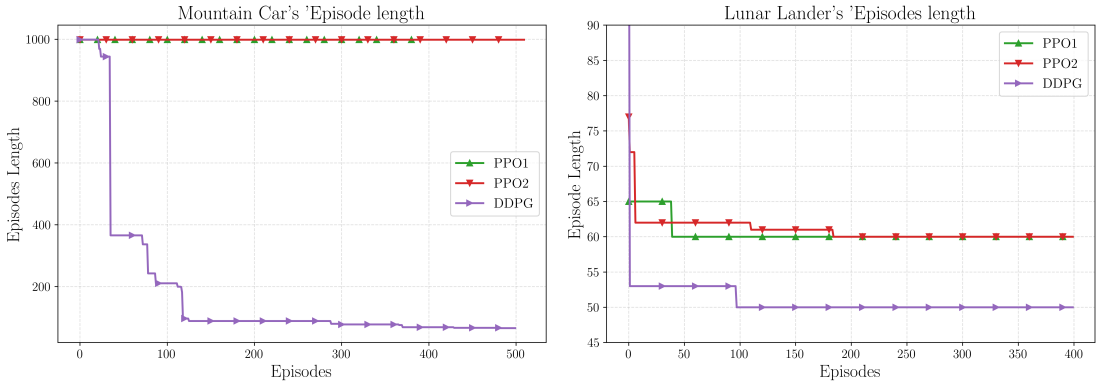


Figure 3.4: Episode's length by learning for Mountain Car and Lunar Lander

process.

For Mountain Car continuous the training ended after 1000 episodes if the cart does not reach the flag on the top of the mountain. This episode's feature is evident in Figure 3.4 for PPO1 and PPO2, in fact on the correspondent learning curves graph (Figure 3.3) PPO1 and PPO2 do not reach to learn enough given that their maximum cumulative reward is the lowest. On Mountain Car we can see effective learning progress via DDPG and the two optimizers: in DDPG, especially, we can observe an impressive decrease in episode length while the other two ANN based algorithms do not perform this behavior. Here we can realize the strength of DDPG among the ANN-based algorithm!.

Similar things could be said for Lunar Lander, though the episode does not end when it reaches a certain amount of steps, for which the episode length continuously varies.

For the total time-steps for learning in the stable baselines' tools, we have put the value $5 \cdot 10^5$; downline of the previous considerations we can assess that the total

episodes used to learn change with the algorithm. In Mountain Car, DDPG can learn faster than PPO# so the episode length needed to finish the episode, and reach the flag in this case, decrease continuously. This is why we can assess that for both PPO algorithms the episodes used in the training process are 500 (that is 5e5/1000 where 1000 is the default steps number to conclude the episode) while DDPG uses more episodes (5e5 steps is a user-defined constant so if the learning process is faster, the episode end earlier and for the same total steps more episode could be evaluated). An algorithm festers than another has a double positive effect due to (i) its velocity itself, in terms of episodes, to reach the maximum and (ii) the possibility to search the objective function's maximum on a wider episode amount. Downline the simulations' data acquired we can draw some evident conclusions: the two optimizers, LIPO and BO, perform the best reward achievement, and they are very fast in that. When we talk about the velocity used to achieve the maximum cumulative reward, we mean the velocity in terms of episodes number because, as we can notice in table **??**, BO is not properly "fast" in sense of the time but just the opposite.

Chapter 4

Application on O.D.E. system

The previous tests have shown how a specified model works and for that purpose, the environments used were game's environments as instances.
Right now we can approach a new environment that describes the mass-spring-dumper evolution, a more physic environment than before, where we appreciate the control action on a "pendulum".
The environment is composed of one mass, one spring, and one dumper free to move along the x-axes (Figure 4.1). In this one-dimensional problem, we take care of a harmonic forcing function F which can disturb the physic system and it can modify its motion law so that the control actions are required to maintain one designed position.

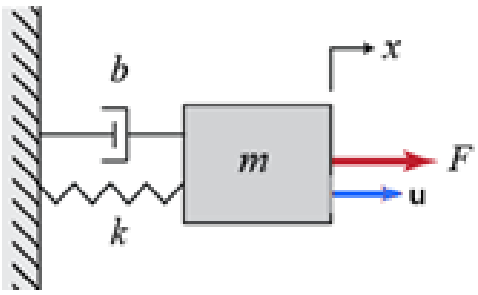


Figure 4.1: ODE system environment

Since the physic problem is 1-D, the physic law describing the mass motion along the horizontal direction involves the forcing function F and the control action u also horizontal.

The mechanical system response is a nonlinear oscillator with one degree of freedom indicated with $x = x(t)$, where x represents the punctiform mass displacement. The nonlinear device shown in Figure 4.1 is based on the generalized Van der Pol damping model. This component alternatively provides and drains mechanical energy from the nonlinear oscillator leading to a limit cycle in the dynamical behavior.
By using the analytical methods of classical mechanics, the nonlinear equation of motion of the mechanical oscillator can be readily written as follows:

$$F_i + F_d + F_k + F_f + F_u = 0$$

Where the first term F_i is the inertia force, the second term F_d is force generated by the nonlinear damper, the third one F_k is the spring force or the elastic response, the F_f is the forcing function while the last one term F_u concern the control actions. Now we can explicitly rewrite the system equation of motion as:

$$m\ddot{x}(t) + \dot{x}(t)(ax(t)^2 + b)c + kx(t) = f(t) + u(t, x, \dot{x})$$

where on the right are all the external forces while on the left only the physic system response.
The values $m = 1\ kg$, $k = 2\ kg/s^2$, $c = 0.3\ kg/s$, $a = 1\ m^{-2}$, $b = -1$ are constants and these are respectively the mass, the elastic coefficient and the three coefficients for the non-linear term of the dumping force. The forcing function is a harmonic function that is only dependent on time while the control force is dependent mainly on the position and velocity of the mass exactly like a closed-loop control. This is common use in reinforcement learning applications because the actions are dependent on the environment state observed, as explained at first in this document.
In the solving process, the mathematical approach is the one for an initial value problem (IVP) for Ordinary Differential Equations (ODE). The solving process is iterative along the temporal variable and in each time step, the system is solved given the state information, position, and velocity, at the previous time step. Precisely the system to be solved is the following:

$$\begin{cases} \ddot{x} = -\frac{1}{m}\left[\dot{x}c(ax^2 + b) + xk - f(t) - u(\dot{x}, x)\right] \\ \dot{x} = \dot{x}(t = 0) \\ x = x(t = 0) \end{cases}$$

In this way we can control the system time evolution for every time step finding the weights whose best fit the null displacement; the flat solution, the aim of the control process, that is the null displacement.

4.1 Control on O.D.E. system

For a recap, the main reinforcement learning components at each step are:

- observation state S: point mass position and its velocity so that $S = [x, \dot{x}]$;
- action: the action definition depends on the algorithm: into the optimizer, the action is defined by the user as a linear combination among state and weights ($W = [w_1, w_2]$) so that $action = W \cdot S^T$, while DDPG and PPO use linear combination through ANN and GP finds the best equation for the state vector components;
- reward: is defined as the absolute difference of the local displacement x and the null displacement, hence the absolute value at the current instant;
- episode length: 40 seconds.

In the previous chapter, we talked about the possibility to create our own environment starting from the general format proposed by OpenAI GYM. The ODE environment has been built exactly similar to the OpenAI GYM environments hence it contains the main interaction functions used for both Mountain Car and Lunar Lander environment, which are: "step()" and "reset()".
Given this environment we can use the same algorithm used before to optimize and control the new environment, never seen before but with the same characteristic. In this context, the LIPO, BO, DDPG, PPO, and GP algorithms will find the best weights given the observation state for the system at stake, completely similar to the previous environments.

Here the control's effect is shown for the forced and unforced case, where the forcing function is the harmonic function $sin()$ and the initial condition is null velocity and displacement equals 4 m.

$$\begin{cases} \ddot{x}(t) + 0.3\dot{x}(t)(x(t)^2 - 1) + 2x(t) = 100\sin(2\pi \cdot 0.4 \cdot t) + u(t, x, \dot{x}) \\ \dot{x}(t = 0) = 4 \\ x(t = 0) = 0 \end{cases}$$

where has been used a non-dimensional forcing functions' frequency of 0.4. Obviously, when we treat the unforced ODE system the forcing term is null.

Looking at the forced solution in Figure 4.2, LIPO and BO appear to perform the same control, but the cost for that is not the same.
Based on what is shown in the table 4.1 for the not forced system, we can conclude that LIPO is faster than BO and it reaches the best reward.
Delving into the weights vector in same table, we can observe the maximum weights value inside it: in fact, running the optimizer, we have set the investigation domain for the weights between -150 and 150, so that we can try to increase the weight's

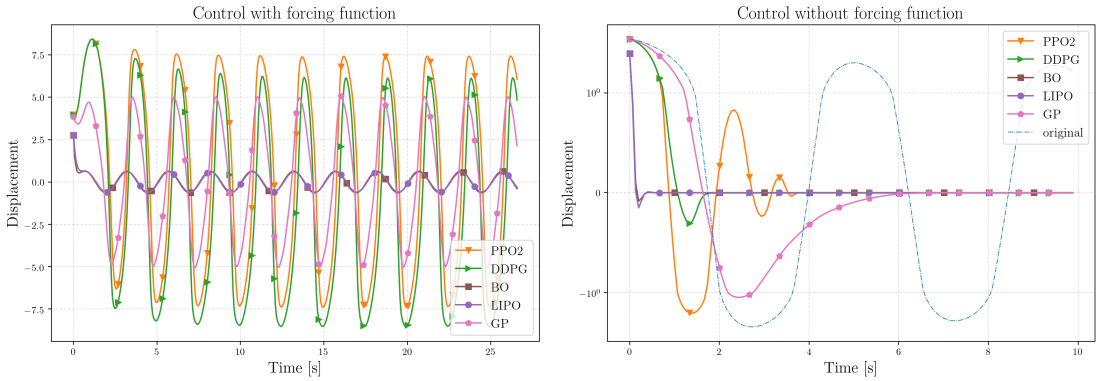


Figure 4.2: Control with different algorithms (with and without forcing function)

	Not forced		Forced	
	BO	LIPO	BO	LIPO
Episodes	200	2000	200	2000
Time [min]	3.77	3.21	7.35	4.60
Reward [m]	0.6348	0.6085	0	0
Weights	[-150,-18]	[-150,-17.47]	[-150,-23]	[-149.64, -29.98]

Table 4.1: LIPO and BO performances comparison

domain expecting to obtain a better reward, but the scope in this section is still to understand the algorithms' behavior.
In the case without forcing function, the control actions are made to "kill" the natural response for the mass-spring-dumper system. In this last case, LIPO and BO attain the best performance killing the whole oscillations almost immediately, without transients, but the two optimization algorithms have different abilities in doing that. Their best weights are very similar, in fact in the learning curves table (Figure 4.3) we can find the corresponded curves almost overlapped. In the case of learning curves practically overlapped we can conclude that the BO and LIPO perform the same behavior but LIPO is a bit faster.
This conclusion about LIPO performances is acceptable when the weights' domain is little but it could be not true otherwise.

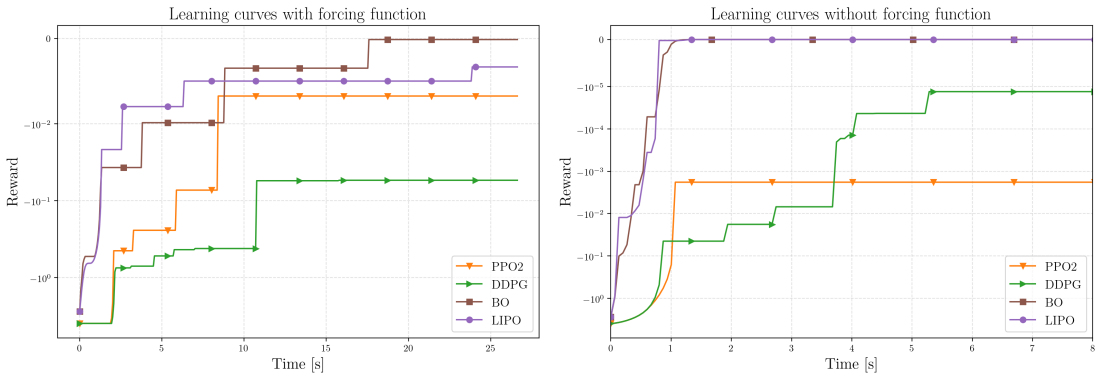


Figure 4.3: Learning curves along the episode for different algorithms

Chapter 5

The BLEW environment

We now approach the actual problem of this work: the control of liquid film undulation.
The control of a liquid film undulation implies the capability to interact with it; for this purpose, a proper environment has been modeled at Von Karman Institute so that the user can simply interact with it managing its outputs via the algorithm he wants.
In this first section, we are going to introduce the reader to the physic model used to build the wiping nozzle for the liquid film interaction environment.
The Integral Boundary Layer (IBL) models are widely used for *falling liquid films* description because, when these are referred to as low dimensional models, eliminate the velocity and pressure fields from the full set of Navier-Stokes equations reducing the number of variables that govern the problem, thus describing the dynamics of the liquid film only through the thickness and the flow rate. These models have been largely used in more complex configurations due to the fact that them enable minor computational cost, if compared to full simulations, and introduce useful analytic insights. As reported in [14], the classical Integral Boundary Layer (IBL) models for liquid films have been extended for a flow configuration that has never been used: the *jet wiping* process. This process on stack involves the use of an impinging gas jet to control the thickness of a coating liquid film on a vertical moving substrate, and it is characterized by unstable dynamics. In particular, the propagation of instabilities of the gas jet to the impinged liquid produces a non-uniform coating distribution referred to as "undulation".
The phenomena described in BLEW concern the 2D formulations. The modeling of this configuration presents two distinctive features: the upward motion of the vertical substrate and the simultaneous presence of time-dependent sources of shear stress and pressure gradient. These formulations aim at predicting the final coating thickness by describing the mean thickness distribution of the liquid film just under the action of the pressure gradient and the shear stress produced by

the jet impingement. The formulation presented neglects the role of inertia in the liquid film, disregarding the nonlinear contribution of advection.

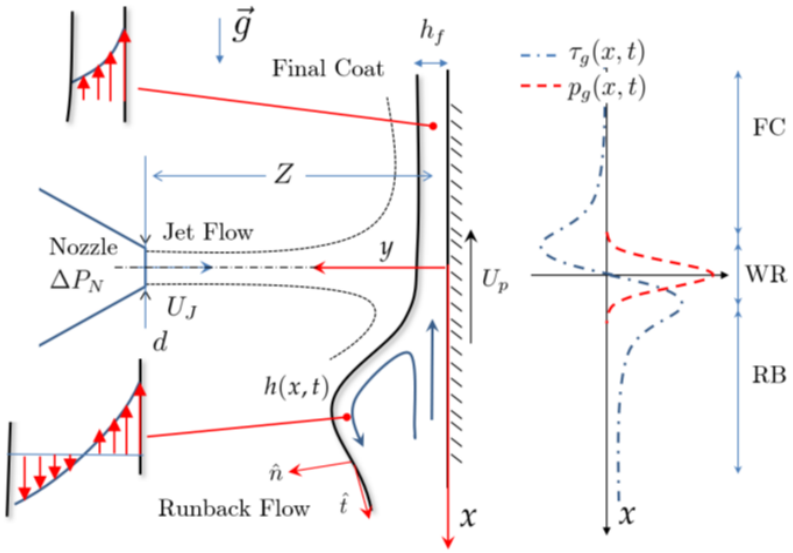


Figure 5.1: Process description [14]

A gas jet impinges on a liquid film that is dragged along a vertical plate moving upward at a constant velocity U_p.
The wiping nozzle is a nozzle with an opening d and a stagnation pressure $_N$ releases a jet flow at a distance Z from the dip-coated substrate moving at a speed U_p. The impingement produces a wiping meniscus (region WR). This forces the film to flow backward (region RB) and leaves a thinner liquid film downstream (region FC) before solidification takes place.
The configuration is assumed two-dimensional, with incompressible liquid flow bounded by the laminate at $y = 0$, and the liquid interface at $y = h(x,t)$; this is a dynamic interface.
The nozzle axis marks the origin $x = 0$, and the streamwise coordinate x follows the direction of gravity, countering the substrate velocity. The impinging jet flow produces a pressure $p_g(x,t)$ and a shear stress distribution $\tau_g(x,t)$ that identifies three areas, qualitatively pictured on the right (Fig 5.1). In the wiping region indicated as "WR", the pressure gradient imposed by the gas jet forces part of the liquid to move in the opposite direction of U_p, resulting in a wiping meniscus. The falling liquid forms the runback flow in the region $x \to$ (indicated as RB); the remaining liquid evolves upward in the final coating region $x \to -$ (indicated as FC). In a coupling formulation has been assumed that the presence of the liquid film does not influence the gas jet and this has been extensively validated for the prediction of the averaged final coating thickness. [15] We thus assume that both

the pressure and the shear stress produced by the jet solely depends on the nozzle gauge stagnation pressure ΔP_N, the nozzle opening d, its discharge coefficient C_d and the standoff distance Z.
All the integral models investigated rely on the Navier-Stokes equation and the related boundary conditions in the "long-wave" formulation. This formulation is derived by scaling the cross streamwise direction with a reference length $[h] = \nu_l U_p/g)^{\frac{1}{2}}$, which is much smaller than the streamwise reference length $[x] = [h]/C_a^{\frac{1}{3}}$. The dimensionless continuity and the momentum equation in the x and y directions, for the long-wave formulation of the problem, reduce to the boundary layer equations:

$$\frac{\partial \hat{u}}{\partial x} + \frac{\partial \hat{v}}{y} = 0, \tag{5.1}$$

$$\varepsilon Re \left(\frac{\partial \hat{u}}{\partial \hat{t}} + \hat{u}\frac{\partial \hat{u}}{\partial \hat{x}} + \hat{v}\frac{\partial \hat{u}}{\partial \hat{y}} \right) = \frac{\partial \hat{p}_l}{\partial \hat{x}} + \frac{\partial^2 \hat{u}}{\partial \hat{y}^2} + 1 \tag{5.2}$$

$$0 = \frac{\partial \hat{p}_l}{\partial \hat{y}} \tag{5.3}$$

Hereinafter the dimensionless variables scaled with respect to the quantities in Table 5.1 are indicated with a hat (e.g. $\hat{h} = h/[h]$).

Reference Quantity	Definition	Expression	$(\cdot)$ reference
$[h]$	$(\nu[u]/g)^{1/2}$	$(\nu_l U_p/g)^{1/2}$	cross streamwise length
$[x]$	$[h]/\varepsilon$	$[h]/C_a^{1/3}$	streamwise length
$[u]$	U_p	U_p	moving speed
$[v]$	εU_p	$U_p C_a^{1/3}$	cross stream velocity
$[p]$	$\rho_l g[x]$	$(\mu_l \rho_l g U_p)^{1/2}/C_a^{1/3}$	liquid pressure
$[\tau]$	$\mu_l[u]/[h]$	$(\mu_l \rho_l g U_p)^{1/2}$	liquid shear stress
$[t]$	$[x]/[h]$	$(\nu_l/U_p g)^{1/2}/C_a^{1/3}$	problem time scale

Table 5.1: Reference quantities for the Shkadov-like scaling

With the film parameter $\varepsilon = C_a^{1/3}$ whose $C_a = \mu U_p/\sigma$ is the capillary number dependent to the surface tension σ we can define $Re = [u][h]/\nu_l = (U_p^{1/3}/g\nu_l)^{1/2}$ is the global Reynolds number of the phenomena. Since the proposed scaling laws hold for $\varepsilon \ll 1$, the long wavelength formulation is valid for $C_a^{1/3} \ll 1$. This generally occurs in galvanizing conditions, where typically $\mu_l \approx 0.003Pa \cdots, \sigma \approx 0.8N$ and $U_p = 1 - 2m/s$ and hence $C_a \approx 0.004 - 0.008$.

The kinematic boundary conditions at the wall and the gas-liquid interface set:

$$\begin{cases} (\hat{u},\hat{v}) = (-1,0) & in\ \hat{y}=0 \\ \hat{v} = \frac{\partial \hat{h}}{\hat{t}} + \hat{u}\frac{\partial \hat{h}}{\hat{x}} & in\ \hat{y}=\hat{h} \end{cases} \tag{5.4}$$

The dynamic boundary conditions at the interface simplify to:

$$\hat{p}_l|_{\hat{h}} = \hat{p}_g(\hat{x},\hat{t} - \frac{\partial^2 \hat{h}}{\partial \hat{x}^2}) \quad along\ \hat{\mathbf{n}}\ in\ \hat{y}=\hat{h} \tag{5.5}$$

$$\frac{\partial \hat{u}}{\partial y}\bigg|_{\hat{h}} = \hat{\tau}_g(\hat{x},\hat{t}) \quad along\ \hat{\mathbf{t}}\ in\ \hat{y}=\hat{h} \tag{5.6}$$

The integral model reduces the modeling complexity reducing the problem 1D.

$$\frac{\partial \hat{h}}{\partial \hat{t}} + \frac{\partial \hat{q}}{\partial \hat{x}} = 0 \tag{5.7}$$

$$\varepsilon Re\left(\frac{\partial \hat{q}}{\partial \hat{t}} + \frac{\partial \hat{\mathcal{F}}}{\partial \hat{x}}\right) = \hat{h}\left(1 - \frac{\partial \hat{p}_g}{\partial \hat{x}} + \frac{\partial^4 \hat{h}}{\partial \hat{x}^4}\right) + \Delta\hat{\tau} \tag{5.8}$$

where $\hat{q}$ is the volumetric flow rate per unit width, and

$$\mathcal{F} = \int_0^{\hat{h}} \hat{u}^2 d\hat{y} \tag{5.9}$$

$$\Delta\hat{\tau} \equiv \hat{\tau}_g - \hat{\tau} = \hat{\tau}_g - \frac{\partial \hat{u}}{\partial \hat{y}}\bigg|_{\hat{y}=0} \tag{5.10}$$

are the advection and the shear stress terms respectively where τ_w is the wall shear stress.
To close integral models, the velocity profile is assumed to be the superposition of three terms:

$$\hat{u}(\hat{x},\hat{y},\hat{t}) = \hat{u}_F(\hat{x},\hat{y},\hat{t}) + \hat{u}_C(\hat{x},\hat{y},\hat{t}) + \hat{u}_P = \hat{u}_F(\hat{x},\hat{y},\hat{t}) + \hat{\tau}_g(\hat{x},\hat{t})\hat{y} - 1 \tag{5.11}$$

The term $\hat{u}_F$ accounts for the contributions of gravity, viscous stresses, surface tension, and pressure gradient. The term $\hat{u}_C = \hat{\tau}_g\hat{y}$ accounts for the shear stress produced at the gas-liquid interface while $\hat{u}_P = -1$ accounts for the motion of the substrate. This kinematic decomposition satisfies the boundary conditions if $\hat{u}_F = 0$ at $\hat{y} = 0$ and $\frac{\partial \hat{u}_F}{\partial \hat{y}}$ at $\hat{y} = \hat{h}$.

5.0.1 Laminar Film Model

The term $\hat{u}_F$ is decomposed in a series of basis function as:

$$\hat{u}_F = \sum_{j=0}^{N} a_j(\hat{x},\hat{t}) f_j\left(\frac{\hat{y}}{\hat{h}(\hat{x},\hat{t})}\right), \tag{5.12}$$

where a_0 is the highest order $O(1)$ and a_j with $j > 0$ are corrections of order $O(\varepsilon)$. The basis functions f_j are taken as:

$$f_j = \bar{y}^{j+1} - \frac{j+1}{j+2}\bar{y}^{j+2}, \tag{5.13}$$

where $\bar{y} = \hat{y}/\hat{h}(x,t)$ is the reduced coordinate. This choice was introduced for falling liquid films, imposing that each of the basis functions satisfies the boundary conditions.

By the assumption that only the first $j = 0$ term of the velocity profile is relevant, we derive the 0-Order Film Model. This means that N = 0 in (5.12) and that the inertial effects can be neglected, that is $\varepsilon Re \sim 0$ and the LHS in (5.8) vanishes. Therefore this model is an inertialess model and the velocity profile became parabolic:

$$\hat{u} = a_0(\hat{x},\hat{t})\left[\left(\frac{\hat{y}}{\hat{h}}\right) - \frac{1}{2}\left(\frac{\hat{y}}{\hat{h}}\right)^2\right] + \hat{\tau}_g\hat{y} - 1 \tag{5.14}$$

As for the 0-Order model, the Integral Boundary Layer (IBL) considers only the first term in the velocity profile (5.14), but it does not assume that the LHS of (5.8) vanishes. The full expansion in (5.12) is now considered for the Weighted Integral Boundary Layer Model (WBL), taking up to N = 4 terms. This number can be derived based on the order of magnitude analysis. The method of weighted residuals to derive the coefficients $A = a_0,\ a_1,\ a_2,\ a_3,\ a_4$ in (5.12) for a falling liquid film was proposed by [16]. In the formulation, a model following the polynomial matching procedure that involves neither weights nor residuals is used because this allows deriving an explicit equation for the coefficients a_j.

Laminar integral boundary layer models have been proved successful for cases up to $Re \approx 80$ well above their theoretical range of validity, while much higher Reynolds numbers, like those considered in this work, might need a different treatment. A different approach is commonly encountered in the literature on shallow water flows in which a correlation between the wall shear stress and a shape factor for the velocity profile is introduced. The Transition and Turbulent Boundary Layer (TTBL) model for the jet wiping problem extends the IBL model to a turbulent liquid film. A similar extension of the WIBL for a falling liquid film, using the mixing-length formulation, is presented by [17].

5.0.2 The Wiping Actuator

The gas jet action's effect on the liquid film is modeled via the pressure gradient and the shear stress produced on the liquid surface. These two quantities, referred to as wiping actuators, are modeled via experimental correlations for gas jet impinging

on the plate, with minor adaptations to account for their time dependency, and under the assumption that the dynamics of the liquid film has no influence on their evolution. The wiping actuators are of the form:

$$\frac{\partial \hat{p}_g}{\partial \hat{x}} = \frac{\partial}{\partial \hat{x}}[P_g(t) f_p(\tilde{x}(t))/(\rho_l g)] \tag{5.15}$$

$$\hat{\tau}_g = T_g f_\tau(\tilde{x}(t)/\sqrt{\mu_l \rho_l g U_p}) \tag{5.16}$$

where $\hat{x}$ denotes a time-dependent axis accounting for the possible oscillation of the jet. The functions f_p and f_τ have range $\in [-1,1]$ so that the maximum values from these quantities are defined by the scalars $P_g(t)$ and $T_g(t)$. Following the empirical correlation, the pressure distribution for a gas jet impinging on a wall is:

$$f_p(\xi) = exp(-0.693\xi^2) + \frac{0.01895|\xi|}{1 + (\xi - 1.67489)^2} \tag{5.17}$$

where $\xi = x/b$ is a dimensionless coordinate, with the parameter b controlling the spreading of the distribution. As proposed by [18], for a stand-off distance $Z/d > 5$, this parameter can be computed as $b = 0{,}125Z$. The maximum pressure P_g, for a statistically stationary impinging jet, can be computed as $P_g = 6{,}5P_d d/Z$, where $P_d = C_d \Delta P_N$ is the dynamic pressure at the nozzle outlet and C_d is the discharge coefficient taking into account the losses for friction and gas separation in the nozzle chamber. The parameter C_d depends on the nozzle design and is taken as $C_d = 0.8$. For more details, the reader is referred to [19].
Concerning the dynamic evolution of the actuators, the time dependency, two possibilities are considered: pulsations and oscillations. In the case of pulsations, the amplitude of the actuators $P_g(t)$ and $T_g(t)$ are set as harmonics with mean value equal to the correlations in steady-state conditions and amplitude of 30%. In the case of oscillations the amplitudes are left stationary and equal to the correlations previously proposed, while the streamwise variable is taken as $\tilde{x}(t) = \tilde{x} - Ztan(W(\theta(t)))$, where $\theta(t)$ is the angle of the oscillation with respect to the horizontal, taken as $\theta > 0$ for a jet deflected upstream (on the runback flow side), and $W(\theta(t))$ is a possible waveform of the oscillation.
Three waveforms are considered. The first is a harmonic oscillation $W(\theta(t)) = \theta_A sin(2\pi f_h t)$. The others are non-harmonic oscillations biased upstream or downstream. These time dependent jet perturbations were designed to mimic different oscillatory modes in the impinging jet flow.

5.0.3 Numerical Methods

We here introduce the numerical methods to solve the set of equations (5.7) . This is a system of hyperbolic PDEs that can be written in the general form:

$$\frac{\partial \mathbf{V}(x,t)}{\partial t} + \frac{\partial \mathbf{F}(x,\mathbf{V})}{\partial x} = \mathbf{S}(x,t,\mathbf{V}) \tag{5.18}$$

with $\mathbf{V}$ the state vector of the problem, $\mathbf{F}$ the conservative flux and $\mathbf{S}$ the source term.
In both the IBL and the WIBL model, the state vector is $\mathbf{V} = [h, q]^T$, the flux term is $\mathbf{F} = [q, \mathbb{F}\beta]$ and the source term is:

$$\mathbf{S} = \begin{bmatrix} 0 \\ (\hat{h} + \hat{h}\frac{\partial^3 \hat{h}}{\partial \hat{x}^3} - \hat{h}\frac{\partial \hat{p}_g}{\partial \hat{x}}) + \Delta\hat{\tau} \end{bmatrix} \frac{\beta}{\varepsilon Re} \tag{5.19}$$

where the coefficient $\beta = 6/5$ only for the WIBL model and $\beta = 1$ otherwise.
The system of PDEs in 5.18 has been extensively treated in the literature on non-homogeneous Shallow Water (SW) equations and a wide range of suitable Finite Volume (FV) schemes for their numerical analysis. Among these, two major classes can be distinguished in the literature: (i) methods arising from Godunov's scheme and so based on the (approximated) solutions of the Riemann problem and (ii) methods arising from the Lax-Friedrich scheme and so based on centered fluxes. The first class of methods is better suited to cope with strong gradients while the second has a much lower computational cost. Because the investigated simulations do not produce shocks within the space and time domain of interest, this work focused on the second class of methods.
Centered schemes introduce a certain amount of artificial viscosity which can be introduced by a suitable combination of low order schemes and high order schemes. This combination is achieved using flux limiters to blend a high order scheme (Lax-Wendroff) in regions where the solution is smooth with a low order scheme (Upwind or Lax-Friedrich) in regions which present strong gradients.
This approach combines the advantages of the two options: first-order schemes prevent numerical oscillations (dispersion) at the cost of excessively smoothing the solution.
A standard Finite Volume (FV) formulation using explicit methods with a three-point stencil in conservative form discretizes 5.18 as:

$$\mathbf{V}_i^{k+1} = \mathbf{V}_i^k - \frac{\Delta t}{\Delta x}\left[\mathbf{F}^+ - \mathbf{F}^-\right] + \Delta t \cdot \mathbf{S}_i^k \tag{5.20}$$

where $\mathbf{F}^+ = \mathbf{F}(\mathbf{V}_i^k, \mathbf{V}_{i+1}^k)$ and $\mathbf{F}^- = \mathbf{F}(\mathbf{V}_i^k, \mathbf{V}_{i-1}^k)$ are the fluxes on the right and the left boundaries of each cell. In a flux limiting scheme, these are

$$\mathbf{F}_i = \mathbf{F}_i^H + \left(\mathbf{F}_i^L - \mathbf{F}_i^H\right)\phi_i \tag{5.21}$$

where ϕ_i is the flux limiter, $\mathbf{F}^H$ is the flux calculated from a high order scheme, and $\mathbf{F}^L$ is the flux calculated from a low order scheme. The chosen limiter function could be the classical min-mod:

$$\phi_i = max(0, min(1, \theta_i)) \tag{5.22}$$

or the Van-Albada otherwise:

$$\phi_i = \frac{\theta_i^2 + \theta_i}{\theta_i + 1} \tag{5.23}$$

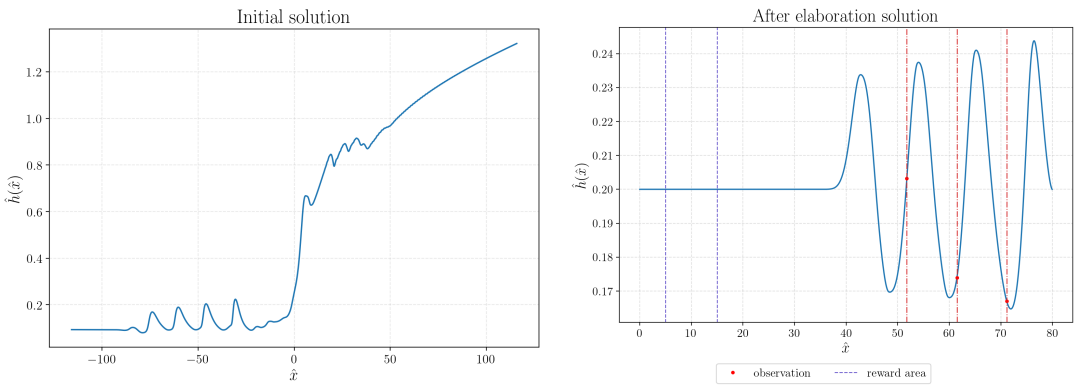
where $\theta_i = (h_i - h_{i-1})/(h_{i+1} - h_i)$ is the smoothness parameter based on the liquid film thickness.
The numerical diffusion added by the low order scheme results in the smoothing of the waves in the liquid film. This smoothing becomes more evident as the waves move away from the wiping point that engenders them. Lax Wendroff scheme is used when the solution requires a high accuracy but its computational cost is high too instead of the Lax-Friedrich method that is less accurate and its computational cost is lower; the numerical diffusion added by this last method introduces a numerical dumping effect that is inappropriate in control problem due to the convergence effect to the mean value of oscillations.

5.1 Time improving from whole phenomena to ROM

How easy to understand, one of the problems in BLEW (Boundary LayEr Wiping) software is the time required to process an entire episode. For the episode here we mean the number of steps required by the model solver to observe two wave period over the reward zone. The reward, the observation, and the active zone are defined by the user and this parameter could change as we wish.
The time per episode is paramount in our research because we use reinforcement learning algorithms that process hundreds of episodes to evaluate the best policy; in this work, it becomes the most important feature because we want to understand how the algorithms interact with the physic model to control the undulation phenomena.
In the beginning, the model was designed to represent the whole wiping phenomena in a wide spatial range, where we can observe all the zinc liquid film exceeding the wanted thickness value wiped in the opposite direction of the laminate in the direction "$-U_p$" (Figure 5.2a). The original episode for this kind of environment is 128 steps long (a bit more than three periods on the reward zone) and it requires a bit more than 1 hour to compute it completely.
In our case the control of the liquid film thickness is necessary to decrease the undulation, hence observing the whole wiping phenomena is not relevant for our purpose so the control environment could regard only the zone where the undulation phenomena appear. If we think in these terms, the wiped liquid flow exceeding the wanted thickness value is not important and a Reduced Order Model (ROM) is created under the shadow of the initial one; this is a ROM which contains only the

undulation waves in its resolution. In accord to [14], trying to represent the wiping effect on a quiet liquid film, the ROM environment shows the solution for a liquid film with no shear stresses on the basement and a forcing function in the origin; this last one concern only the mass flow and not the thickness properly, in fact, the thickness is keeping constant while the mass flow rate change via a harmonic forcing function (Figure 5.2b).

This trick allows to save almost 55 minutes of almost 1 hour of the original episode



(a) **(b)**

Figure 5.2: BLEW output in complete solution and ROM

128 steps long so that after this tweaking the episode takes almost 7 minutes.
The time performance increasing is amazing, from 1 hour to 7 minutes, but it is still not enough; just for one hundred episodes an algorithm still needs approximately 12 hours. In order to decrease the episode computing time-consuming, we focus more on the design environment so that the space interval length changes from 150 to 80 non-dimensional points, and has been decided to observe only two oscillation periods on what we call the reward zone concerning the definition of the episode length. The time required for an episode is 5 minutes; it is still too much.
Understanding the model computation process becomes evident that the most time requirement for an episode is taken by the initial transitory; actually, we do not care about those first steps because the related first undulations are far from the wiping control nozzle and the times used in this transitory is lost waiting for the undulation approach the control nozzle area. Starting with a different initial condition became necessary to decrease the episode time required, so an initial condition with the transitory already developed and more steps already computed has been decided to use as the initial condition.
Moreover, we define the episode length as the length needed to observe only two undulation wavelengths onto the reward zone and this sets a limit in our simulations.
After these precautions, the model requires only 2 minutes to perform 86 steps,

required for the definition of episode length, in a space discretized through a medium-mesh (1318 points in 80 non-dimensional distance points): now it seems we do not have any other possibilities to decrease the time required for an episode.

5.1.1 The step length

Waiting for the discussion on the mesh type and the flux limiter used for the simulations, we try to explain how the episode length is affected by the step length used for evaluating the “instant by instant" solution. In order to obtain the thickness amplitude on the mesh points, the equations discussed before are solved each “step", but the time-length of this step is decided by the user. Previously has been assessed that for an episode (two-wavelength observed onto the reward zone), 86 steps were required; each of these 86 steps contains inside it 50 minimum numerical time-steps used for solving the differential equations.
To avoid confusion in the nomenclature we step back into what we define as “time-step".
Originally, in the chapter concerning the GYM's environment, we talked about the possibility to create the our own environment starting from the general format proposed by OpenAI GYM. The BLEW environment has been built exactly similar to the OpenAI GYM environments hence it contains the main interaction functions used for both Mountain Car, Lunar Lander, and the ODE system environment, which are the famous: “step()" and “reset()".
Since we use the “step()" method to evaluate the action on the model's solution, we can put inside this step a user-defined amount of numerical time-steps used in the discretized differential equations for the wiping model.
For a recap here we define the:

- numerical time-step: the minimum time used discretization of the wiping model's differential equations and defined as the minimum representative time among the gas and the liquid phenomena: in this case is $dt = 0.01667s$ for the liquid;
- step: the amount of numerical time-steps used to perform one “step()" and along which the action is kept constant.

Since the user can pick an arbitrary amount of numerical time-steps into each step, we could choose an enormous quantity on those to minimize the time required to process an episode, but if we extend this idea to its limit we could have the faster-computed episode with only one step, so one single action along all the episode. This strategy is yet not useful for control purposes because it only aims to decrease the episode time computing but we can expect the worst control performance; the best choice, for control purposes instead, is to have an actuator with the possibility to interact with the liquid film each numerical time step, the smallest time-step, but

this increases enormously the episode time computing. When we call the solution "instantly", we mean the solution while the actuator produces action on the liquid film surface every 0.01667 seconds instead of the usual 50 times larger step is 0.833 seconds (when the episode was 86 steps long each step was 0.833 seconds long). This multiple is chosen as big as to ensure the fast conclusion of an episode, but we can lose the accuracy in acting because the actuator could choose an instant in a time domain 50 times smaller for its actions. Making smaller and smaller the step toward the numerical time-step the time required by the computer to complete the episode increases: the instant episode computing end after 291 seconds instead of the 50 times bigger step which requires 143 seconds, less than the half part, but with lower time accuracy. As said before, become evident the necessity to mediate between the temporal accuracy and the episode computing time required.

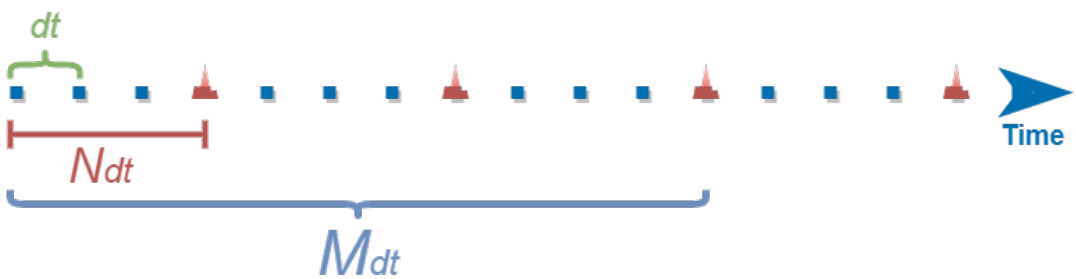


Figure 5.3: The numerical time-step dt, the step N_{dt} and episode M_{dt}

5.1.2 The CFL condition

The numerical time-step is the same for all the mesh types because the CFL condition is not implemented yet in this environment. Knowing the characteristic time for the liquid and the gas phenomena, the numerical time-step used for solving the differential equations is chosen as the smallest between these two characteristic times.
In each step, a small amount of numerical error is collected into the solution hence different waves shape could be observed for different mesh types. Throughout the episode, only the numerical diffusion effect on the spatial resolution due to the mesh refinement could be observed on the waves, but not the relative time diffusion due to the CFL condition is nonexistent. Mainly the Lax-Wendroff scheme is the most problematic scheme since when the mesh becomes finer (from Fine Mesh on) its numerical solution pours on NaN values earlier and earlier in time increasing the mesh fineness. Via comparison of the four mesh types, this effect becomes visible; only the solution's space distribution is dependent on the mesh grid but observing their peaks in Figure 5.4 we can notice that at the beginning and at the

end of an episode the space distribution maintain the same distance among the different solutions, so there is not mesh dependence in numerical time-step, the CFL condition is not actually implemented.

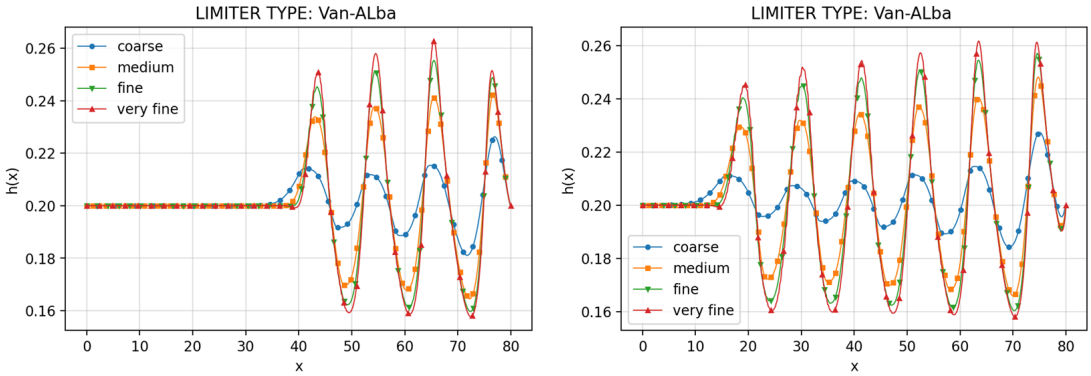


Figure 5.4: proof of absence of CFL condition

5.2 Flux Limiters effects on model's solution

Lax Wendroff scheme is used when the solution requires a high accuracy but its computational cost is high too instead as Lax-Friedrich method which is less accurate and its computational cost is lower; the numerical diffusion added by this last method introduces a numerical dumping effect that is inappropriate in control problem due to the convergence effect to the mean value of the oscillation how we can see in 5.5.
How we could expect the Lax-Wendroff method, the higher-order method, to provide a more numerical accurate solution while the Lax-Friedrich method, the lowest order method, goes to introduce the highest dumping effect. The Flux limiters are very able to reduce the negative numerical diffusion effect but the time to process an episode increases for the lowest order solver (let see Table 5.2). The time difference required by each episode with one or other Flux limiters could appear insignificant and our final choice could be obviously direct to the Lax-Wendroff solution, the most accurate. From an operational point of view, we have to match among time required and solution accuracy because the time needed to process an episode becomes noteworthy as soon as the optimization run; along the hundreds of episodes required for the optimization, the time saved is hundreds of times the small difference for each episode and whereas the episode length increase also the small time difference increase. In general, this trend is visible when the mesh refinement increases. Therefore, it could be very useful to

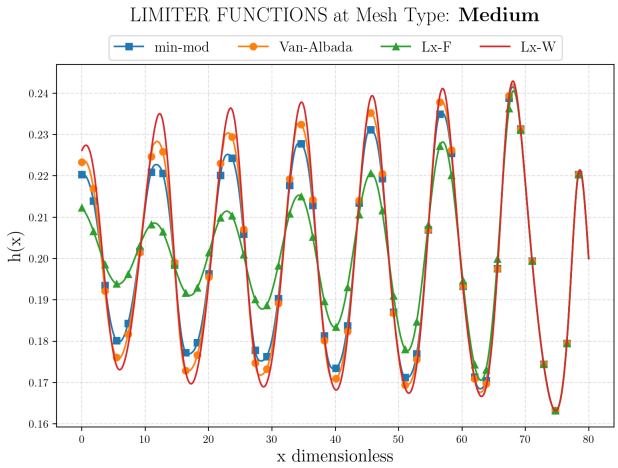


Figure 5.5: Limiter Functions' solution

strive for Van-Albada Limiter because of a good compromise between the time required and numerical accuracy.
Due to numerical stability matters, the limiter's ϕ_i is filtered on the space domain and this produces a non-uniform distribution of ϕ over the space. This numerical requirement causes the solving of both the low order and high order methods for each step for both the hybrid method (min-mod and Van-Albada) and the Lax-Wendroff method. However, the Lax-Wendroff method requires more times for an episode than the other methods due to the spatial filter because the limiter is calculated and it is not taken into account as constant ($\phi(x) = 1$); the time required increases especially when the mesh grid goes to be finer and finer, as for the other limiters.

Time required for one episode [s]					
Limiter Type	Coarse Mesh	Medium Mesh	Fine Mesh	Ultra-Fine Mesh	Extreme Mesh
Min-mod	80	126	176	233	279
Van-Albada	76	126	180	228	227
Lax-F	75	125	182	226	273
Lax-W	75	127	194	238	277

Table 5.2: Time required for an episode

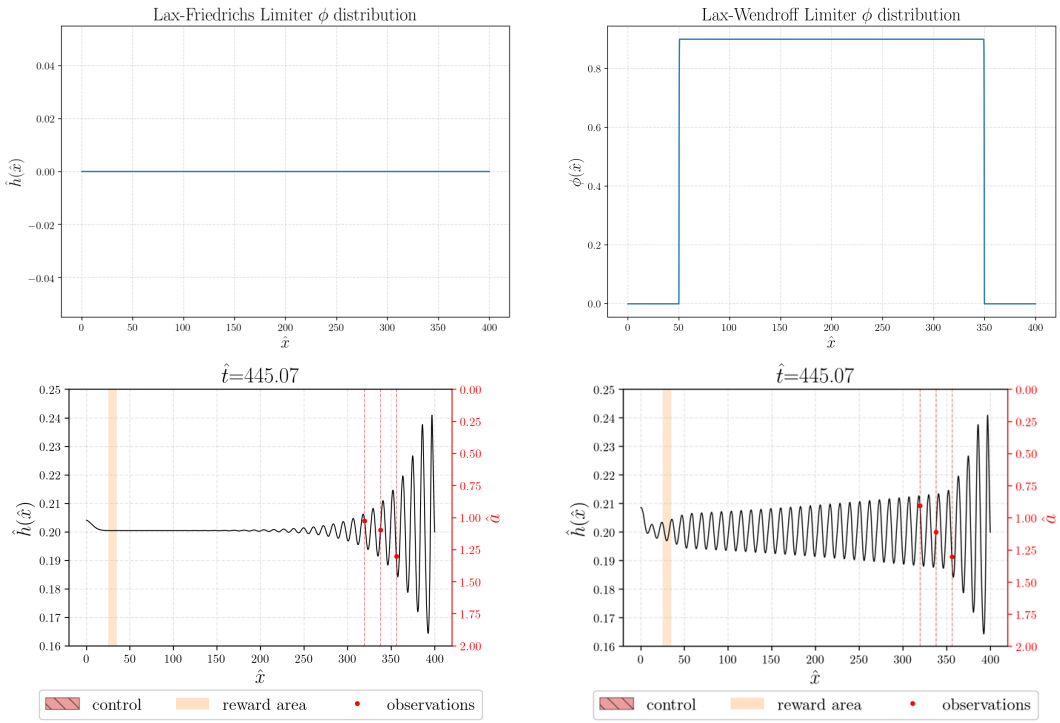


Figure 5.6: Flux limiter distribution and their solution accuracy

Chapter 6

The jet wiping control on BLEW

Let analyze mainly the influence of the number of jets on the controlled solution after hundreds of episodes through the algorithms exposed. We will understand how much important could be the episode length and, obviously, the policy used in the training processes.

6.1 The single jet control

Working on the BLEW's parameters which could influence the learning process in the RL algorithms, the step length treated in the previous chapter represents the most important data. Successively is going to be reported the algorithms achievement with the two different step lengths: the numerical time-step and the relative 50 times bigger step. Before proceeding with the use of the BLEW environment the algorithms have been improved by tweaking the ANN size or other hyperparameters; the main change concerns the size of the ANN in DDPG and PPO# while for the optimizer other hyperparameters are used as constant in the next optimization processes.
We report the new ANN enlarged to obtain the best weights combination along the learning process. The PPO# algorithm, proposed by Stable Baselines, uses a Multi-layer perceptron (Mlp) consisting of two layers of 64 perceptrons using ReLU as activation function; the new ANN for PPO# is still a "Mlp" consisting of fifteen layers of different dimension for 32 perceptrons per layer to 512 as shown in the Figure 6.1.
The Table 6.1 could be included the new ANN for policy (π,"pi") and value function (vf) which input is:

```
policy_kwargs = dict(net_arch=[dict(
                pi=[64, 128, 128, 256, 256, 512, 512, 512, 256, 256,
                                                128, 128, 64, 32],
                vf=[64, 128, 128, 256, 256, 512, 512, 512, 256, 256,
                                                128, 128, 64, 32])],)
```

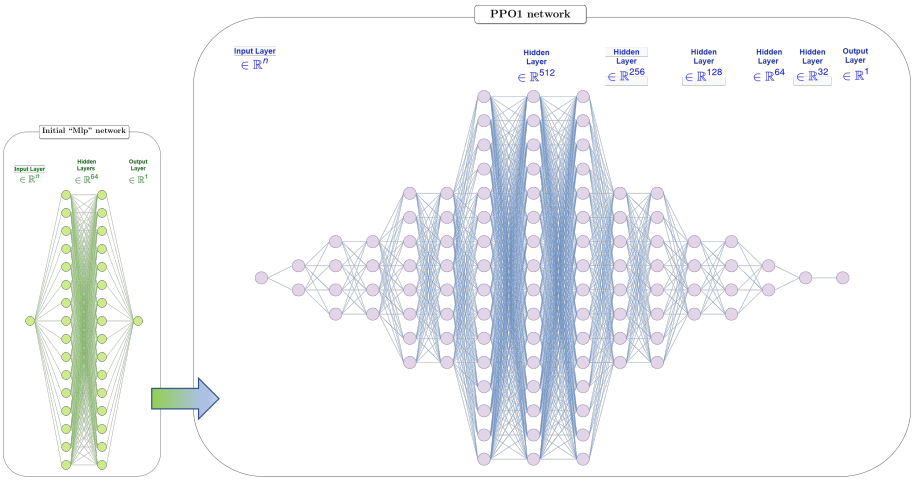


Figure 6.1: PPO2 ANN improvement

Similarly to PPO the DDPG also used a Multi-layer perceptron (Mlp) consisting of two layers of 64 perceptrons using ReLU as activation function, but we tweaked it to an Mlp consisting of seventeen layers from 32 to 512 perceptrons per layer as shown in Figure 6.2. The new ANN is related to the policy only.

```
policy_kwargs = dict(layers=[64, 128, 128, 128, 256, 256, 512, 512,
    512, 256, 256, 128, 128, 128, 64, 32])
```

In DDPG(H) we can be more deep specifing every sub-network as in Figure 6.3. In the case of the Optimizer like LIPO and BO, the main hyperparameters are shown in Table 6.1. At moment we do not include Genetic Programming in our discussion because only step length influence on the whole performance is going to be discussed.
As regards the reward we have chosen the Gaussian distribution of standard deviation for the boundary condition, this last one represents the target to achieve, the mean value of liquid zinc thickness h. This is what is recommended in [20]: using the Gaussian distribution of the original reward leads the algorithm to a faster learning process. The reward is defined as the sum over the steps of the standard deviation with respect to the target value, the mean value, for the solved

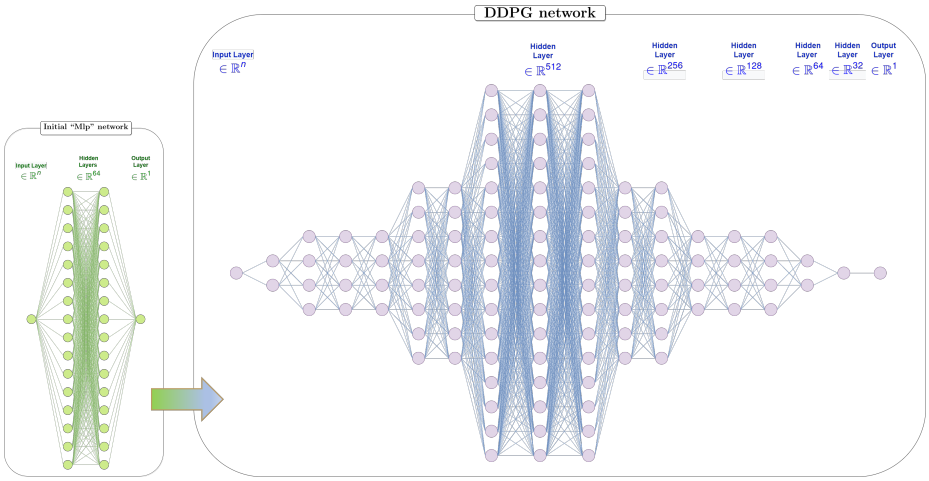


Figure 6.2: DDPG ANN improvement

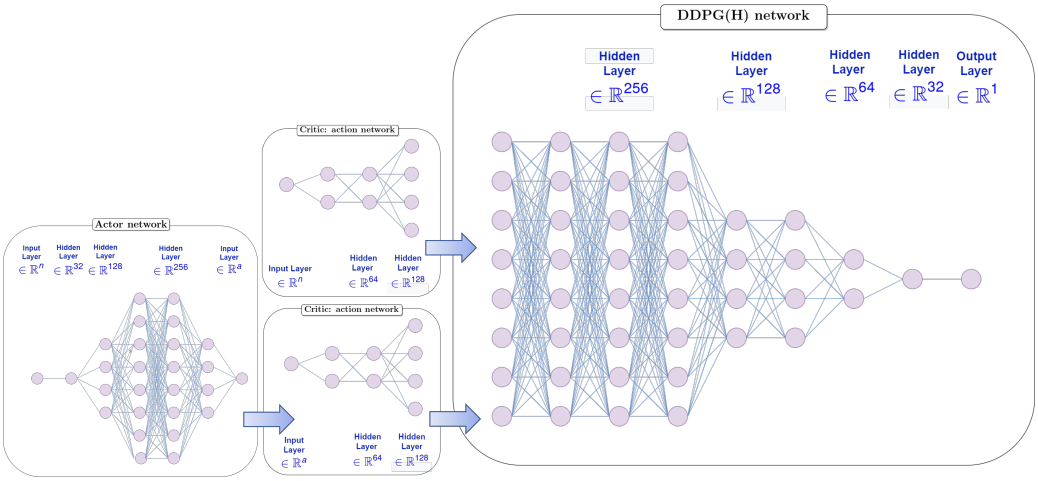


Figure 6.3: Inside the DDP(H) networks

solution along the time variable; this value is used as a negative exponent for an exponential function, so that: e^{-std}. The reward definition will be discussed more afterward in order to analyze its faults (Section 6.4).
The rewards achieved are dependent on the step length because the step is processed outside the solving process. The final reward is the sum of the reward obtained every step, so whereas the step is composed by 50 numerical time-steps the output will be a number 50 times smaller than the reward obtained when the step is the numerical time-step. To be clear, the user chooses the step length and after this step, the reward value is given as output and will be summed along the episode; if the episode uses 86 steps to end then the total reward will be the sum of 86

LIPO	
Initial seed points:	500 points
Initial point generator:	random
Weights set:	{-15,15}

BO	
Initial points:	1000 points
Initial point generator:	random
Weights set:	{-15,15}
Acquisition function	Expected Improvement

PPO1	
Steps per actor update:	3 episodes
Optimizer's batch size:	two-thirds of episode
Optimizer's epochs number:	3
lambda :	0.93

PPO2	
Steps per actor update:	3 episodes
Optimizer's batch size:	two-thirds of episode
Optimizer's epochs number:	3
Optimizer's mini-batches number:	3

DDPG

Instantly step	
Buffer size:	1 million steps
Batch size:	two-thirds of episode

50 numerical steps	
Buffer size:	20 thousands steps
Batch size:	two-thirds of episode

Table 6.1: Algorithms' parameters for BLEW

immediate rewards but when the user chooses to use the smaller step, the numerical time-step, the episode will require 4200 steps and the total reward will be the sum of 4200 immediate rewards.
The total reward scaled according to its episode length is called the "immediate average reward". These are the rewards shown in Table 6.1 and in Figure 6.4 which are the best reward achieved through the learning process and scaled with respect to the episode length!
The reward has been set on LIPO, DDPG and PPO as a negative reward because there are some maximizers instead of BO which is a minimizer and his reward has been set as positive. The last note for LIPO and BO concerns the seeding points: they are randomly assigned because if we send out the points on an equally spaced grid along the weights domain the algorithm could lose time optimizing the results into local maxima.
In Table B.1 and Figure 6.4 all the simulations are carry out using a mesh grid of Medium dimension and the Van-Albada flux Limiter.
As expected, the learning curves for instant actions show higher rewards than the others because the action is constant only for one numerical time-steps and then it could change instead of maintaining it constantly for 50 numerical time steps. The algorithms achieve a better reward because the actions can be better distributed along the time to fit better the target solution. We need to choose the step length equal to the numerical time step to obtain the best reward but the time required to complete the training is too much. The final decision was taken downstream this

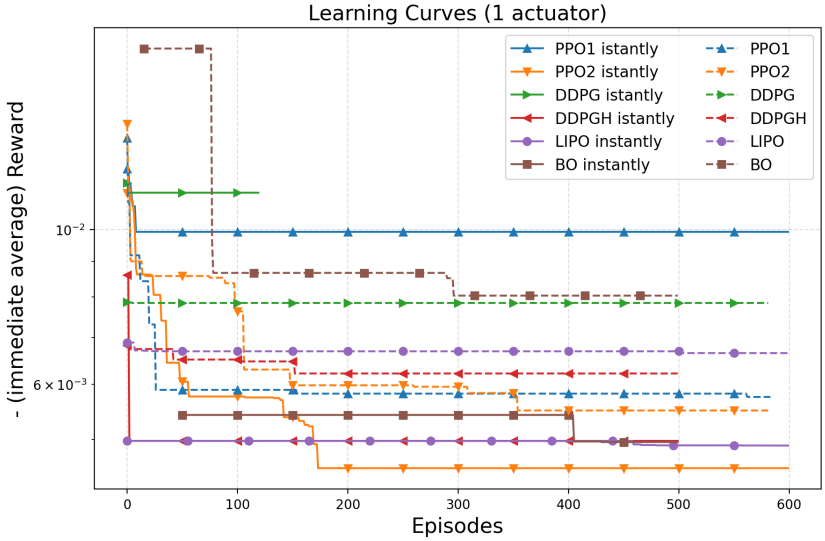


Figure 6.4: Performances differences in negative rewards

first analysis is about using a step length smaller than 50 times the numerical time steps, but it is still not the smallest one. For the next simulations and training will be used a step length of 20 numerical time-steps which ensures actions release more frequently and maintained constant enough less than before but it also ensures an episode time required smaller than the 4200 steps episode, almost the 50% more.

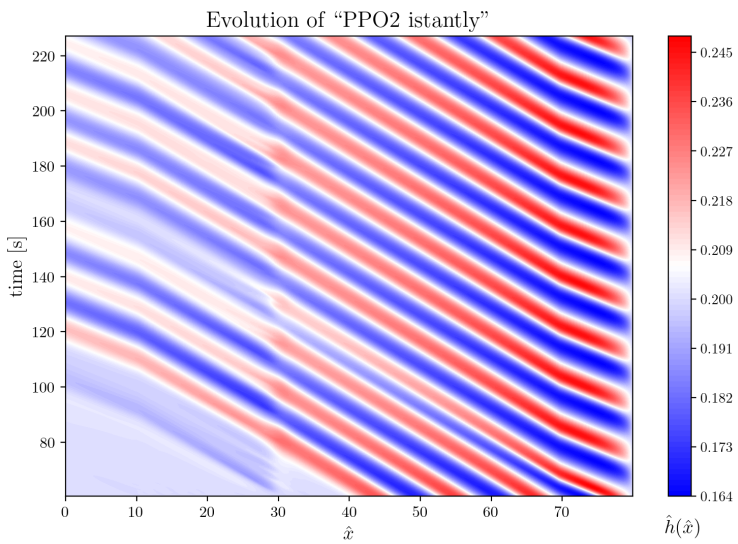


Figure 6.5: PPO2 instantly evolution

The best solution for instantly PPO2 is shown in Figure 6.5. There we can notice how the undulation's peaks are recognized and wiped becoming almost white, and flat, after the control action on $\hat{x} = 30$ but the blue parts, the meniscus, are still evident and they are not enough controlled. The wiping nozzle can only blow and it cannot aspire, so the meniscus parts will be maintained unless the wiping control nozzle will be inclined to wipe the liquid zinc toward the meniscus. In the next learning processes the actuator number will increase to two actuators expecting a better controlled solution.

6.2 The double jets wiping control

In parallel to the previous optimizations has been developed the double jet control on the liquid film introducing also different policy laws for the optimizer LIPO and BO and tweaking more parameters. Here the previous conclusion was not already known and the simulations have been run using the faster method, so the step() is 50 times the numerical time-step.
In the algorithms, the parameters are maintained almost the same as previously because they are reported in Table 6.1 as proportional to the episode length used. The principal differences are located in the policy for the optimizer LIPO and BO. The new policies still manage the weights in a new linear combination but also use them in an open-loop control via harmonic function.
For first, the new linear policy has been designed to examine the undulation shape: the undulation's waves peak is located at the beginning, the dimensionless x = 80, and its maximum value, when the Lax-Wendroff limiter is used, is less than 0.25. The wiping nozzle can only blow until the maximum or blow out giving information, the action, included in $0, \ldots, 1 \in \mathbb{R}$. Knowing the actions are related to the observations state, and given the maximum observation value equals 0.25, we can limit the weights domain space using only sums in the linear combination: we can assess, in a certain instant t: $\mathrm{a}=\mathrm{w}_1 s_1 + w_2 s_2 + w_3 s_3 \;\Rightarrow\; 0.25(w_1 + w_2 + w_3) = 1$ *so that* $w_i = \frac{4}{3}$.
We will use only sums in linear combination, so the weights will be only positive to ensure the maximum action=1. In this way the research domain of weights will definitely decrease and we can expect a better performance other than faster.
The common linear combination policy uses this time the weights set included in -8,8 because given the last results the best weights are close to the value=4.

With regards to the harmonic policy, two different solution will be proposed.
For harmonic policy we mean that the control actions follow an harmonic function along the time, so here we move away from closed loop control to enter in open loop control due to the actions are not dependant from the observations. The two methods proposed will use the same parameters as weights so that optimize the weights means optimize the harmonic function's parameters. Given the amplitude

$\mathcal{A}$, the non dimensional frequency $\mathbf{f}$ and the phase ψ. the harmonic policy could be one of the following:

$$a_t(w) = \mathcal{A}sin(2\pi\mathbf{t}t + \psi) \begin{cases} \mathcal{A} = & \{x|x \in \mathbb{R} : 0 \leq x \leq 1\} \\ \mathbf{f} = & \{y|y \in \mathbb{R} : -50 \leq y \leq 50\} \\ \psi = & \{z|z \in \mathbb{R} : 0 \leq z \leq 2\pi\} \end{cases} \tag{6.1a}$$

$$a_t(w) = 0.5 + \mathcal{A}sin(2\pi\mathbf{f}t + \psi) \Longleftarrow \begin{cases} \mathcal{A} = & \{x|x \in \mathbb{R} : -0.5 \leq x \leq 0.5\} \\ \mathbf{f} = & \{y|y \in \mathbb{R} : -50 \leq y \leq 50\} \\ \psi = & \{z|z \in \mathbb{R} : 0 \leq z \leq 2\pi\} \end{cases} \tag{6.1b}$$

In equations 6.1 the weights are relative the parameters $\mathcal{A}$, $\mathbf{f}$ and ψ, each can wander inside a particular domain space.

The training for Genetic Programming (GP) is also evaluated for double jets control after changing the algorithms in order to output two actions instead of only one because here we use two actions instead of one. This tweak implicate more time to finish the optimizations but it is required in this new environment. The parameters inserted into the algorithms concern the number of individuals into the initial population (pop), the probability to mate individuals or crossover (cxpb), to mutate an individual (mutpb) and the wanted offspring (λ) for a given parents (μ) selected, in addiction to the number of generations (ngen): these are the followings in Table 6.2.

Genetic Programming parameters

	pop	cxpb	mutpb	μ	λ	ngen
Simple	100	0.5	0.7	-	-	10
M,L	100	0.35	0.35	40	120	20
M+L	100	0.35	0.35	50	150	20

Table 6.2: GP parameters for the Simple, (μ, λ) and $(\mu + \lambda)$ algorithms

Downstream the training the “Simple" algorithm produces the best performance and it is the one shown in Figure 6.7

When the training finishes, the learning curves in Figure 6.6 show the best performance. The algorithms LIPO and BO with classic linear policy performs the best achievement. The harmonic policy does not work correctly evidently, but more we can say about the policy based on the new linear combination strategy. The new linear combination strategy for policy could appear less expensive, and surely it is, but it is not able to perform a good result because it considers all the observations equally. The observations are weighted at the same way losing the information

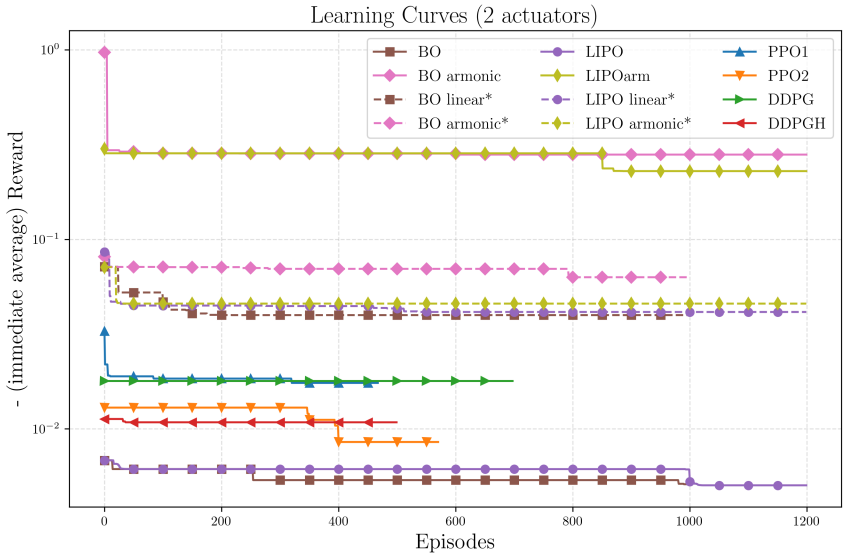


Figure 6.6: Learning curves for double jets control

between the state vector. The weights' domain is too poor to guarantee a good exploration. Among the observations state maybe one could be more representative in dumping oscillation than other and that should have ad higher weight. The best solution is every time to ensure a far exploration trough a wider weights domain as possible to avoid the optimization choke. The weights require to be summed or subtracted as they please giving more or less, positive or negative, contribution in action defining process. The observations' position should change to observe how it influences the search inside the weights domain, but this lie outside the time usable in the current work.
In Figure 6.7 two actuator are used in control but only one is used by the algorithms because the action in one actuator is every time null. This behaviour could be visible also in other trained algorithms so we can affirm that only one actuator is enough to control the liquid film in order to kill the undulation, while the second actuator is redundant: only one actuator will be used in successive optimizations. In figure the undulation's peaks are not completely dumped but the negative aspect are the deep meniscus created by the jets. This behaviour move the solution away from the target solution and we can conclude that the control has not been performed sufficiently good and more episode could be used in more training, especially respect the BO and PPO2 which seems them could improve more the reward achieved. In Table B.2[1] the BO and LIPO weights are very close to the

[1]The symbol " * " specify the "new" version: for harmonic refers to the equation 6.1b

maximum value achievable so the weights' domain space needs to be wider.

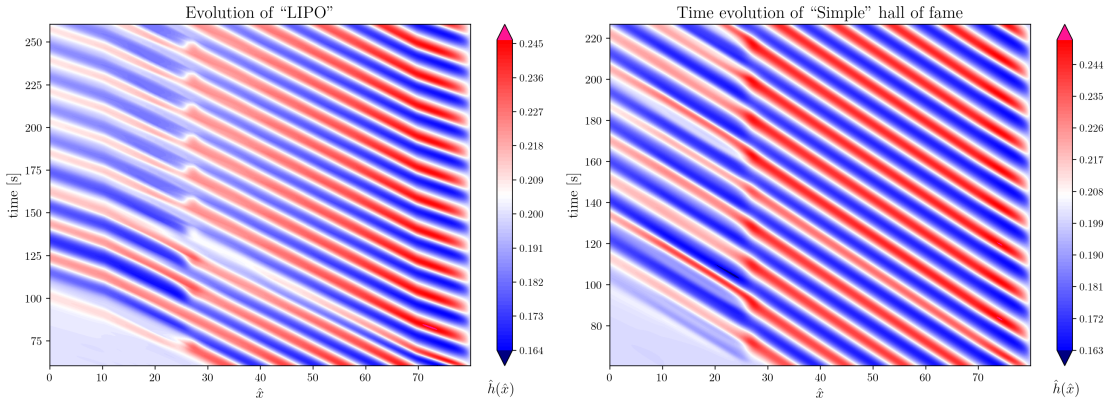


Figure 6.7: LIPO and Simple evolution

From the comparison of the two solutions for the linear LIPO and the Simple evolutionary algorithm for GP become evident the best control is performed by the LIPO; this one presents slightly bright colors after the control which ensure lower peaks and higher meniscus around the mean value 0.2. Also in the Simple evolutionary algorithm is adopted only one actuator but for $\mu + \lambda$ and μ, λ algorithms also the second jet located in $\hat{x} = 40$ has been used but with poor performances.

6.3 The definitive jets wiping control

The last BLEW version used for control purposes includes the expedients shown in the conclusions of the previous sections. The BLEW environment uses now the step-long 20 numerical time-steps, which ensure more precise actions use in time, and only one actuator because it is sufficient to control the liquid film undulation and it also provides faster training than the two actuator environments.
The episode is now 500 steps long but this information will not be significant in our further considerations. The noteworthy information is that before, to complete an episode 4200 numerical time steps were required but now, where the step is composed of 20 numerical time steps, the required steps to complete one episode are 210. Now we use 500 steps because a problem occurs in BLEW control: the reward zone is close to the action zone but they do not coincide, hence the reward computed is not the one relative to the action but it is the reward obtained downstream previous actions. The algorithm entrusts a certain reward to some actions which do not create that reward and we can refer to this problem as a "phasing" problem. In order to try to solve this problem the distance between the actuator and reward zone has been decreased moving the actuator from $\hat{x} = 30$ to $\hat{x} = 25$ and the episode

length has been increased so that the observed waves on reward zone are more than 2 but almost 5. This expedient does not solve the phasing problem between action and the evaluated reward but, relating to the phasing problem, it helps to cope with the "delay" problem. The delay problem is related to the delay problem but it refers to the initial actions only; at the beginning, when the actuator acts on the liquid film the reward computed is not relative to the undulation distribution and we need to wait for some numerical time-steps to have at least the reward relative to the perturbed solution by actions and not relative to the initial undisturbed solution. This pours in a different transitory in which the reward is relative to the undisturbed solution even if the actuator is already acting on the liquid film. Making the episode length longer, the "delay transitory" has less weight on the total reward and at least the reward evaluated concerns the effect of the actions.
Moreover, the reward now is not dependent on the episode length but the BLEW environment has been tweaked to obtain the cumulative reward as sums of instant rewards, per each numerical time-step, so that the total reward will not be scaled by the episode length to have a comparative reward because this is exactly the total cumulative reward.
In this environment on stack, the undulation shape is also different and it has more physical features. The undulation shape proposed now is relative to the solution via CFD analysis and its shape recalls a gaussian distribution in space. As shown in Figure 5.2a the undulation has not the harmonic shape but only peaks and no meniscus in its real solution and CFD confirmed this kind of characteristic. Minimizing the solution difference between the "real" shape given from CFD analysis and a parametrized Gaussian curve, the undulation wave acquires the following distribution:

$$fun_{wave} = e^{-\frac{(t-\tilde{t})^2}{2\sigma^2}}$$

where $\sigma = 2.36934558$ is obtained through the minimization process, t is the current instant in simulation and $\tilde{t} \propto \frac{1}{\hat{f}}$ is the centered time for the undulation time distribution which is relative to the non dimensional frequency $\hat{f} = 0.055$: the gaussian undulation does not go along time t but it evolves along the time in relation to the numerical evolution with reference to $\hat{f}$: $t - \tilde{t}$ is the centered time.
The new environment provides the new ROM with Gaussian solution different than the harmonic distribution as shown in Figure 6.8:
The last combination type for policy in optimization algorithms is the non linear combination. Connecting us to [21], for the given state vector $S_t = [s_1, s_2, s_3]_t$ and the non linear form for policy

$$a_t(S, w) = w_t \; S_t^T + S \; W_t \; S_t^T.$$

the weights will be included into both $w = [w_1, w_2, w_3]$ and $W_t \in \mathbb{R}^{3x3}$ which leads in larger amount of weights and the importance of their combination management.

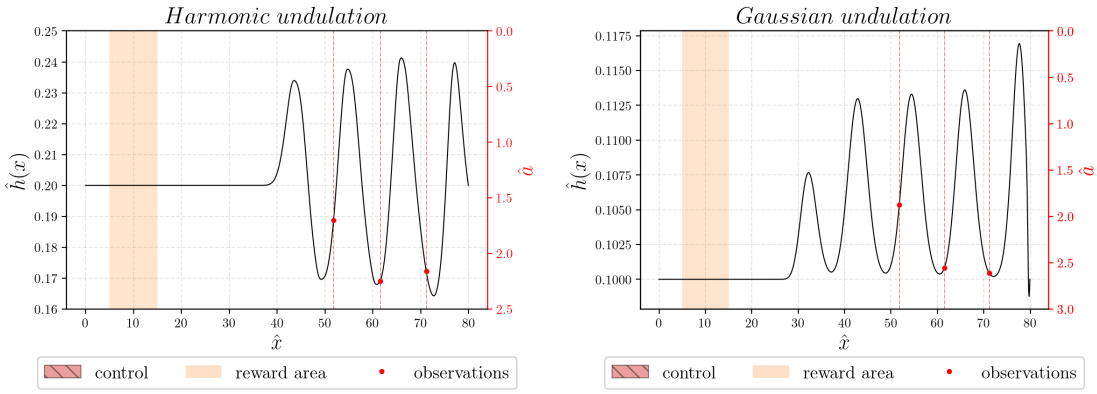


Figure 6.8: From Harmonic to Gaussian undulation in ROM

This last combination will surely require more episodes to achieve the best reward because will be computed the combination of 12 weights, differently than before when there were only 3. The non-linear policy for LIPO and BO could pick one weight inside the set from -18 to +18. After the first attempts became visible the higher performances of LIPO non-linear so has been decided to use another version of this algorithm whose weights' domain space is the set from -50 to +50.
The "LIPO/BO linear" parameters are the ones relative to the new linear policy in equation 6.2 which uses only sums in its definition and positive weights to the maximum value of 1.33; the same discussion could be mirrored for the "LIPO/BO harmonic" whose policy is the new harmonic policy in equation 6.1b.
The GP algorithms now use a new set of values reported in Table 6.3 In (μ, λ) and

Genetic Programming parameters

	pop	cxpb	mutpb	μ	λ	ngen
Simple	100	0.5	0.7	-	-	15
M,L	100	0.35	0.35	25	80	20
M+L	100	0.45	0.45	20	70	20

Table 6.3: GP parameters for the Simple, (μ, λ) and $(\mu+\lambda)$ algorithms in Gaussian undulation

$(\mu + \lambda)$ the probability of mate and mutate individuals are equally but in these cases, the algorithms could also pick some parents for the next generations and not only the offspring; the probability to have a parent in the generative set is given as $1 - cxpb - mutpb$.
The model training performs the learning curves in Figure 6.9 with the features in Table B.3. The LIPO non-linear with larger weights domain attains the best

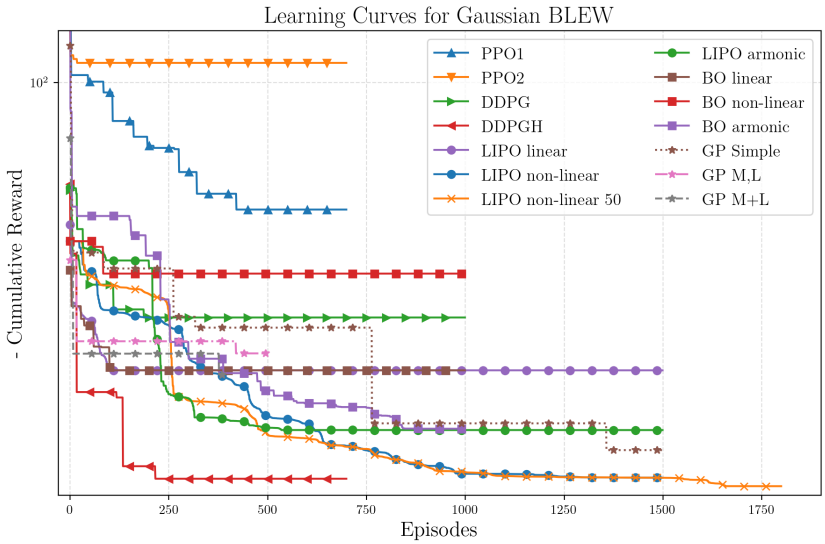


Figure 6.9: Learning curves in Gaussian undulation environment

reward and it appears to have the capability to improve more its performances. We can also notice how the harmonic policy for LIPO and BO performs better than the previous attempt with two actuators, but the best solution is relative to the non-linear policy it is now shown in Figure 6.10.

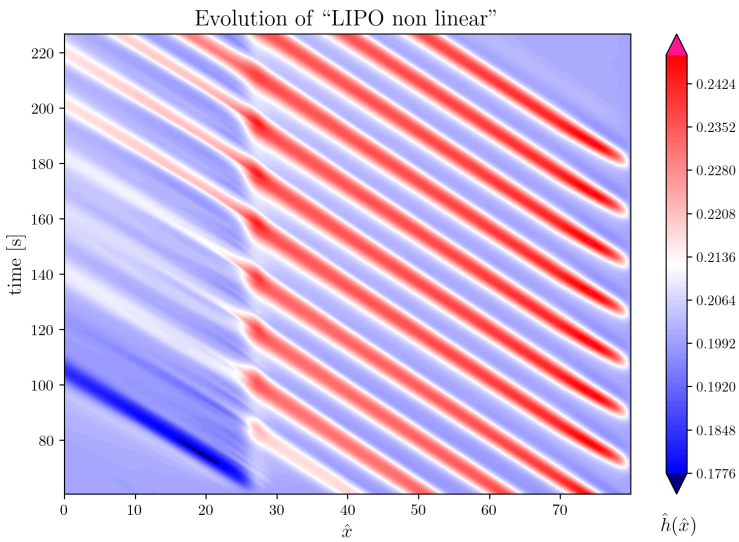


Figure 6.10: LIPO non linear: the best performance

6.4 The reward influence

The main concern in the machine learning algorithm is the reward distribution because the learning velocity is affected by its distribution. Here we try to explain how the undulation on the reward zone affects the convergence to the highest reward peak by using a harmonic function in an arbitrary domain.
The reward in the following examples is defined similarly to the O.D.E. environment, so it is a space difference between the layer thickness wanted and the actual layer thickness in the reward zone. Because the reward zone now is not a single point as in the ODE environment (the reward was the instantaneous space difference at that integration time step), but it is a wider space, we decided to define the reward as a negative standard deviation of the actual solution respect to the initial point value:

$$Reward \equiv -std = -\frac{\sum_{i=0}^{N}(\hat{h}_0 - \hat{h})^2}{N} \tag{6.2}$$

where $\hat{h}_0$ is the wanted thickness value which receives the amount of $d\hat{q}$ which ensure the undulation phenomena in the current ROM model.
Here we also take into account a different reward definition which increases when the points of the solution are closer to the aim. It is the following one:

$$Reward = -std \cdot \left(1 - \frac{\sum_{j=0}^{M} x_j \ \forall x_j : \left|\hat{h}(x_j)\right| < \epsilon}{N}\right) \tag{6.3}$$

where ϵ is an arbitrary small number.
In accord with to [20] a reward distribution with a negative Gaussian function shape could be the best choice to obtain the best result in fewer episodes and smaller time required. The negative reward, generally, ensures avoid being stuck in a kind of loop along the learning process, but in this case, it is more important because the standard deviation is a positive function that decreases when the algorithm is going to approaches the target and maximizes the reward we have to invert the trend. The Gaussian reward distribution helps to flatten the reward when it goes very far from the maximum reward value (the null value), maintaining it almost at a comparable little value while largely increasing when it is in the proximity of the peak: therefore the Gaussian reward distribution helps to focus the algorithm research in the proximity highest value.

$$Reward = e^{-std} - 1 \tag{6.4}$$

If we investigate the reward definition (the standard deviation) we can observe that it is affected by weird behavior. When the oscillation of the target over we are computing the standard deviation has multiple oscillations, its standard deviation

could be equal to the standard deviation for a distribution mean value closer to the target but with fewer oscillations. The last example is the kind of target we are looking for through the ML algorithms but its value could be concealed by another one a bit different. Obtaining a better reward distribution is mandatory to ensure a fast ascending to the global peak and not lose iteration on local maxima.

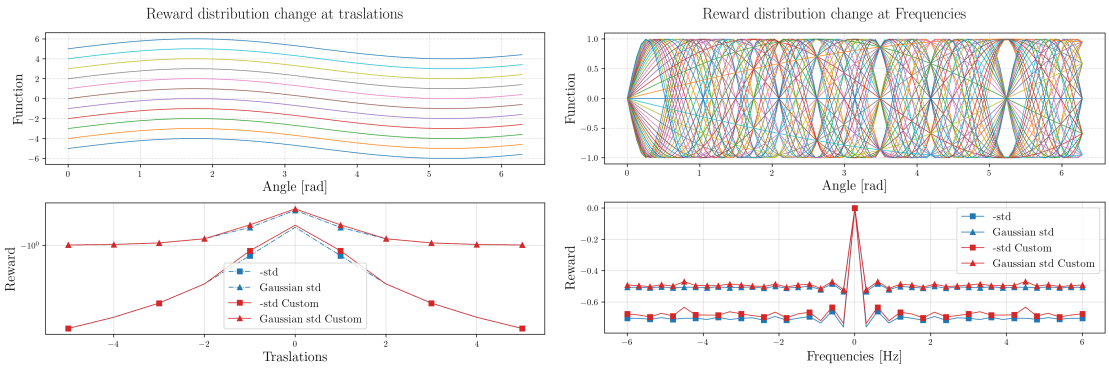


Figure 6.11: Translations and oscillations affects on the reward distribution

Chapter 7

Conclusions and future recommendations

In this paper, the work carried out aimed to show how Reinforcement Learning can benefit engineering applications inherent in closed-loop active control. The control obtained by means of the algorithms now known to this point succeeds very well in damping the characteristic waves of the ripple phenomenon on the liquid zinc film, but as can be seen in the solutions reported, a completely flat solution downstream of the control cannot be obtained.
The algorithms seem to work by completing very good analyses and thus show that they are capable of learning quite effectively. From this, it can be inferred that the suboptimal control, characterized by a liquid layer surface that is not completely flat, is not directly attributable to the algorithm but to the environment in which it acts. Indeed, the BLEW environment has been gradually modified to represent a more physically correct and numerically effective solution for control, but not all precautions have already been implemented. In this work, some precautions have improved the learning speed for some algorithms by affecting the maximum achievable reward but further precautions could make the solution even better.
First, it might be considered a good practice to use episodes of longer duration so that there is a greater chance of learning as early as during an episode, especially for the PPO# and DDPG algorithms that work in just this way. This expedient could also further decrease the weight of the "delay transient" on the whole episode and make the analysis more effective. What has just been said is acceptable to minimize the effects of the error brought by the "delay transient" in our analysis, but in this case, it is allowed in our system. Instead, the best solution would be to eliminate this source of error by placing the reward zone at the action zone so that we have immediate feedback through the immediate reward of the control action being performed.

All optimizations were carried out using a state vector composed of 3 equispaced observations and positioned upstream of the control zone. In this case, the state vector contains information about 3 waves of ripple, and the information about the single wave would be lost. It would be optimal to bring the observations closer together so that we can obtain an action that can better respond to the form of the wave in analysis.

Appendix A

Established concepts

A.1 Artificial Neural Network (ANN)

An ANN is a model inspired by the networks of biological neurons in our brains. Biological neurons produce short electrical impulses called action potentials (or just signals) which travel along the axons and make the synapses release chemical signals called neurotransmitters. When a neuron receives a sufficient amount of these neurotransmitters, it fires its own electrical impulses. They are organized in a vast network of billions, with each neuron typically connected to thousands of other neurons. The architecture of biological neural networks (BNNs) is still subject of research, but starting from the parts of the brain already mapped, it seems that neurons are often organized in consecutive layers. The artificial neuron activates its output when more than a certain number of its inputs are active.
The *Perceptron*, invented in 1957 by Frank Rosenblatt, is one of the simplest ANN architectures and it is based on a slightly different artificial neuron known as the *threshold logic unit (TLU)*. Each input connection is associated with a weight, so the TLU computes a weighted sum of its inputs ($z = w_1x_1 + w_2x_2 + ... + w_nx_n = x^Tw$) and then applies a step function on the sum result to output: $h_w(x) = step(z)$, where $z = x^Tw$.
The most common step function used in Perceptrons is the *Heaviside step function* but sometimes other functions could be instead. A single TLU can be used for simple linear binary classification. It computes a linear combination of the inputs, and if the result exceeds a threshold, it outputs the positive class; otherwise, it outputs the negative class. Training a TLU connected to all the inputs means finding the right values for $w_0, w_1, \ldots, w_n$.
When an ANN of perceptrons is made of multiple layers we call it as Multi-layer perceptron (MLP). When all the neurons in a layer are connected to every neuron of the previous layer, we refer to this layer as a fully connected layer or a *dense*

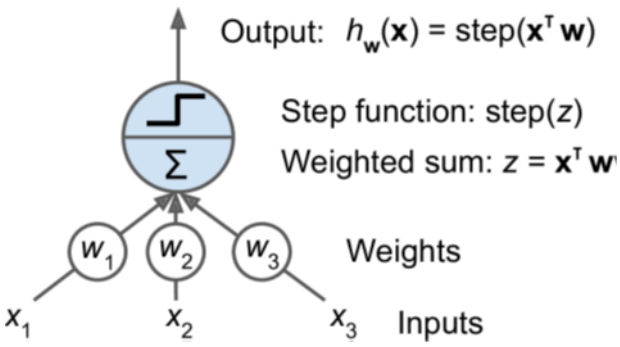


Figure A.1: The *TLU Perceptron*

layer. Moreover, an extra bias neuron could be generally added with the purpose to output the numerical unit all the time, in such a way as to create a constant. Compute the outputs of a fully connected layer means solving $h_{w,b}(X) = \phi(x \cdot w + b)$ where:

- X represents the matrix of inputs;
- The weight matrix w contains all the connection weights except for the ones from the bias neurons which are contained in b;
- The function ϕ is called the *activation function*: when the artificial neurons are TLUs, it is a step function.

Donald Hebb noticed that when a biological neuron triggers another neuron often, the connection between these two neurons becomes stronger. "Cells that fire together, wire together"; that is, the connection weight between two neurons tends to increase when they fire simultaneously. This rule later became known as Hebb's rule. Perceptrons are trained taking into account the error made by the network when it makes a prediction. More specifically, the Perceptron is fed one training instance at a time, and for each instance, it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.

Perceptron learning rule (weight update): $w_{i,j}|_{next\ step} = w_{i,j} + \eta \cdot (y_j - \hat{y}_t)x_i$
- y_j is the output of the j-th output neuron;
- $\hat{y}_t$ is the target output of the j-th output neuron
- η is the learning rate.

When an ANN contains a deep stack of hidden layers, the layers between the input and the output layer, is called a *Deep Neural Network (DNN)*.

Back-propagation is the core of the network training algorithm. It first makes a prediction (forward pass) and measures the error between the prediction and the target, then goes through each layer in reverse to measure the error contribution

from each perceptron's connection (reverse pass), and final tweaks the connection weights to reduce the error. Once it has figured out the errors, it just performs a common Gradient Descent step and the whole process is repeated until the network converges to the solution.
We have also to replace the step function with the *logistic* (sigmoid) *function*, $\sigma(z) = 1/(1 + exp(-z))$. This was essential because the step function contains only constant segments, so the Gradient Descent cannot be applied because it cannot move on a flat surface, while the logistic function, which has a continuous non-zero derivative, allows Gradient Descent to make some progress at every step. Also others functions could be used to fit better the problem solving: these are, as instances, the hyperbolic tangent function $tanh(z) = 2\sigma(2z) - 1$ and the Rectified Linear Unit function $ReLU(z) = max(0, z)$, both with non-zero derivative everywhere.
In general, when building an MLP, we do not want to use an activation function for the output neurons because we want them to be free to output any range of values in a continuous space. Alternatively, we can use the soft-plus activation function, which is a smooth variant of ReLU: $softplus(z) = log(1 + exp(z))$. If we want to guarantee that the predictions will fall within a given range of values, then we can use the hyperbolic tangent, and then scale the labels to the appropriate range.

Here we want to delve into the learning process just announced but is noteworthy to introduce some definitions; in fact, for MLP architecture we define the *batch* as the number of training examples present in a single iteration of evaluation and *epoch* the total training examples, defining the matter on stack, when they pass forward and backward through the neural network once. (As an instance, for a data-set of 1000 examples, 5 iteration steps are required to complete an epoch composed by batches whose batch size is 200 examples).

We review this algorithm in more detail:

I An ANN handles one mini-batch at a time, and it runs the full training set multiple times. Each time the full set is processed, each pass, is called an *epoch*;

II Each mini-batch is passed to the network's input layer and starting from that it runs through all the layers so that the algorithm can compute the output of all the neurons layer per layer (for every instance in the mini-batch). This process represents the *forward pass*: it is exactly like making predictions, except that all intermediate results are preserved since the backward pass;

III Now the algorithm, using a loss function that compares the desired and the actual output, measures the network's output error of the network and returns some measure of the error (*backward pass*).

IV Then it computes how much each output connection contributed to the error. This is done analytically by applying the chain rule (maybe the most fundamental rule in calculus), which makes this step fast and precise.

It is important to initialize all the hidden layers' connection weights randomly because if we initialize all weights and biases to zero, then all neurons in a given layer will be identical, and thus backpropagation will affect them in exactly the same way. In other words, the model will act as if it has only one neuron per layer. The loss function to use during training is typically the mean squared error, but if we have a lot of outliers in the training set and we may prefer to use the mean absolute error instead.

[1]

A.2 Useful tools

To provide the reader with all the useful tools for a clear understanding of the algorithms used in this work, which has been exposed above, we now introduce some numerical methods as the "mainstay" of the algorithms reported. These are nothing else than conditions or minor algorithms widely used, in the sense that they are continuously recalled, in those that are the most known learning techniques reported here.

KULLBACK–LEIBLER DIVERGENCE

The *KL (Kullback–Leibler) divergence*, (also called relative entropy), is a statistical distance and it measures how one probability distribution p diverges from a second expected probability distribution q.

$$D_{KL}(p \mid\mid q) = \int_x p(x)\ log\left(\frac{p(x)}{q(x)}\right) dx \tag{A.1}$$

D_{KL} attains the minimum, the zero, when $p(x) == q(x)$ everywhere.
It is noticeable, according to the formula A.1, that *KL divergence* is asymmetric because in cases where $p(x)$ is close to zero, but $q(x)$ is significantly non-zero, the effect of q is disregarded. This causes buggy results when we just want to measure the similarity between two equally important distributions.
Expressed in the language of Bayesian inference, $D_{KL}(p \mid\mid q)$ is a measure of the information acquired by revising one's beliefs from the prior probability distribution q to the posterior probability distribution p. In other words, the KL divergence takes into account the amount of information lost when q is used to approximate p. Typically p represents the real distribution of data, observations, or a precisely calculated theoretical distribution, while q typically represents a theory, a model, or approximation of p. To find a distribution q that is closest to p, we have to minimize KL divergence.

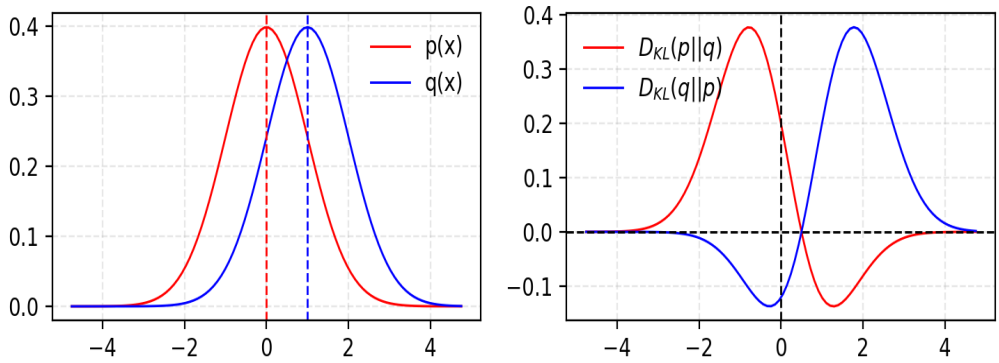


Figure A.2: D_{KL} for p and q as normal distributions

[22]

TRUST REGION POLICY OPTIMIZATION (TRPO)

To improve training stability, we should control the parameter updates to avoid the policy changes too much at one step. Trust Region Policy Optimization (TRPO) [23] carries out this idea by enforcing a *KL* divergence constrained on the size of policy update at each iteration.
TRPO aims to maximize the objective function $J(\theta)$ subject to the *trust region constraint* which enforces the improvement between old and new policies measured by KL-divergence, to be small enough and within a parameter δ:

$$\underset{\theta}{maximize} \quad J_{\theta_{old}}(\theta) \tag{A.2}$$

$$subject\ to \quad D_{KL}^{max}(\theta_{old}, \theta) \leq \delta \tag{A.3}$$

where we can use $\delta = 0.01$ similar than in [23].
Meeting this hard constraint, the old and new policies would not diverge too much, so that TRPO can guarantee a monotonic improvement over policy iteration.

THE POLICY GRADIENT

The policy gradient methods aim to model and optimize the policy directly. The policy is usually modeled with respect to parameters θ, so that could be written $\pi_\theta(a|s)$. The reward (objective) function's value depends on this policy and then we can admit to use various algorithms to optimize θ in order to achieve the best reward.
The objective function (or sometimes, reward function) is defined as:

$$J(\theta) = \sum_{s \in S} d^\pi(s) v^\pi(s) = \sum_{s \in S} d^\pi(s) \sum_{a \in A} \pi_\theta(a|s) Q^\pi(s, a)$$

where $d^{\pi}(s)$ is the stationary distribution of the Markov chain for π_{θ} (on-policy state distribution under π).
Using the gradient ascent, we can move θ toward the direction suggested by the gradient $\nabla_{\theta} J(\theta)$ to find the best θ for π_{θ} that produces the highest return. Due to the fact that $\nabla_{\theta} J(\theta)$ depends on both the action selection (determined by π_{θ}) and the distribution of states following the target selection behavior (indirectly determined by π_{θ}), computing the gradient becomes not immediate and tricky. Moreover, is difficult to estimate the effect on the state distribution by a policy update when the environment is generally unknown. Luckily, the policy gradient theorem could be restated to not involve the derivative of the state distribution $d^{\pi}(.)$ that simplifies the gradient computation $\nabla_{\theta} J(\theta)$.

$$\nabla_{\theta} J(\theta) = \nabla_{\theta} \sum_{s \in S} d^{\pi}(s) \sum_{a \in A} \pi_{\theta}(a|s) \cdot Q_{\pi}(s,a) \propto \sum_{s \in S} d^{\pi}(s) \sum_{a \in A} \nabla_{\theta} \pi_{\theta}(a|s) \cdot Q_{\pi}(s,a)$$

(To go deeply in the proof, see [24])

ADAM OPTIMIZATION ALGORITHM

In using the gradient descent (or ascent) algorithm, the choice of the learning rate, and the step size for progress, becomes crucial in terms of the global performances we are expecting. The learning rate is usually kept constant and defined a priori by the expert user to have good global minima, or maxima, estimation. Its choice affects the computing resource due to the fact that a smaller learning rate requires more steps to attain the minima/maxima and spending some steps in the local minima/maxima, but its approximation will be more accurate. The *Adaptive Movement Estimation* algorithm, or *Adam*, is an extension of gradient descent that automatically adapts the learning rate related to each input variable and furthers an exponentially decreasing moving average of the gradient to update the variables that leads to smoothing the research process.
Adam is indicated on updating many hyperparameters in a similar way to the gradient descent/ascent learning rate and also in problems with very noisy and/or sparse gradients. The method computes individual adaptive learning rates for different parameters from estimates of the first and second moments of the gradients. Adam is designed to accelerate the optimization process decreasing the number of function evaluations required to reach the optimum, or in our case, improving the capability of the optimization algorithm resulting in a better final result. Importantly, each step size is automatically adapted throughout the search process based on the partial derivatives (gradients) for each variable.
Ascribing to m the moving averages estimated for the 1st moment (the mean) and to v the 2nd raw moment (the uncentered variance) of the gradient and given

$\beta_1, \beta_2 \in [0,1)$ the exponential decay rates for the moment estimates, the algorithm works as shown in the following pseudo-code in this section A.2.
Given that the moving averages m and v are initialized as the null vector, leading to moment estimates biased towards zero (especially during the initial time steps), we need to counteract this initialization bias resulting in bias-corrected estimates $\hat{m}_t, \hat{v}_t$.
The problem with ADAM is that it uses hyperparameters but the good new is that the author in [25] recommended default settings for the tested machine learning problems that are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

[25].

Algorithm: Pseudo-code for ADAM

inputs: α: Stepsize
$\beta_1, \beta_2 \in [0,1)$
$f(\theta)$: Stochastic objective function with parameters array θ
θ_0: initial parameter vector

Initialize the 1st moment vector m_0
Initialize the 2nd moment vector v_0
Initialize timestep $t = 0$
while θ_t not converged **do**:
$t = t + 1$
$g_t = \nabla_\theta f_t(\theta_{t-1})$ Get gradients w.r.t. stochastic objective at timestep t
$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) \cdot g_t$ Update biased first moment estimate
$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ Update biased second raw moment estimate
$\hat{m}_t = m_t/(1 - \beta_1^t)$ Compute bias-corrected first moment estimate)
$\hat{v}_t = v_t/(1 - \beta_2^t)$
$\theta_t = \theta_{t-1} - \alpha \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$ Update parameters
end while
return θ_t the resulting parameters.

Appendix B

Tables

Single jet control

		training steps [episodes]	time [h]	reward	max|weight|
Instantly steps	PPO1	3 millions [715]	66	(-)0.0099	-
	PPO2	3 millions [715]	67	(-)0.0045	-
	DDPG	0.5 million [120]	89	(-)0.0113	-
	DDPGH	[500]	96	(-)0.0049	-
	BO	[500]	31	0.0049	2.117
	LIPO	[1000]	23	(-)0.0048	3.099
50 numerical steps	PPO1	50000 [584]	25	(-)0.0057	-
	PPO2	50000 [584]	24	(-)0.0055	-
	DDPG	50000 [584]	25	(-)0.0078	-
	DDPGH	[500]	48	(-)0.062	-
	BO	[500]	15	0.008	3.5
	LIPO	[1000]	34	(-)0.0066	9.9

Table B.1: Characteristics of the learning performance for single actuator control.

Double Jets control

	training steps [episodes]	time [h]	reward	max|weight|
PPO1	40000 [468]	25	(-)0.017	-
PPO2	40000 [571] (bit smaller episode)	25	(-)0.0085	-
DDPG	60000 [698]	40	(-)0.0178	-
DDPGH	[500]	48	(-)0.01	-
BO harmonic	[1000]	126	0.279	40
LIPO harmonic	[2000]	149	(-)0.22	28.49
BO harmonic*	[1000]	97	0.063	20.69
LIPO harmonic*	[1500]	124	(-)0.0457	4
BO	[1000]	38	0.00509	8
LIPO	[1500]	52	(-)0.00503	7.99
BO linear*	[1000]	82	0.039	0.56
LIPO linear*	[1500]	112	(-)0.041	4.89
Simple	-	3	(-)0.0015	-
M,L	-	6.5	(-)0.017	-
M+L	-	3.8	(-)0.018	-

Table B.2: characteristics of the learning performance for double actuator control

Single Jets control on Gaussian

	training steps [episodes]	time [h]	reward	max|weight|
PPO1	[700]	30	(-)63	-
PPO2	[700]	30	(-)107	-
DDPG	[1000]	45	(-)47	-
DDPGH	[700]	25	(-)36.15	-
BO harmonic	[1000]	39.6	0.279	1.01
LIPO harmonic	[1500]	62	(-)39.5	5.28
BO	[1000]	45	43.65	1.33
LIPO	[1500]	61	(-)43.65	1.18
BO non-linear	[1000]	48	50.8	17.8
LIPO non-linear	[1500]	62	(-)36.2	2.65
LIPO non-linear 50	[2500]	90	(-)35	43.1
Simple	-	54	(-)38	-
M,L	-	41	(-)44	-
M+L	-	50	(-)43	-

Table B.3: Characteristics of the learning performance for single actuator control in Gaussian undulation shape

Bibliography

[1] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras and TensorFlow*. OREILLY, 2019 (cit. on pp. 11, 76).

[2] John Schulman. «Proximal Policy Optimization Algorithms». In: (2017) (cit. on p. 13).

[3] C. E. Rasmussen and C. K. I. Williams. «Gaussian Processes for Machine Learning». In: (2006), p. 266 (cit. on p. 14).

[4] Charles E. Clark. «The greatest of finite set of random variable». In: *Operation Research* 9.2 (1961), p. 19 (cit. on p. 16).

[5] Peter I. Frazier. «A Tutorial on Bayesian Optimization». In: (2018) (cit. on p. 16).

[6] M.J.D. Powell. «The NEWUOA software for unconstrained optimization without derivatives». In: (2004), p. 42 (cit. on p. 17).

[7] *A Global Optimization Algorithm Worth Using*. URL: `http://blog.dlib.net/2017/12/a-global-optimization-algorithm-worth.html` (cit. on p. 19).

[8] Nicolas Vayatis C´edric Malherbe. «Global optimization of Lipschitz functions». In: (2017) (cit. on p. 19).

[9] *Genetic Programming*. URL: `https://deap.readthedocs.io/en/master/tutorials/advanced/gp.html` (cit. on pp. 22, 25).

[10] David B. Fogel Thomas Back and Zbigniew Michalewicz. *Evolutionary Computation 1. Basic Algorithms and Operators*. 2000, p. 376 (cit. on p. 25).

[11] *Gym-openAI*. URL: `https://gym.openai.com/` (cit. on p. 27).

[12] *global_function_search*. URL: `http://dlib.net/optimization.html#global_function_search` (cit. on p. 32).

[13] *skopt.Optimizer*. URL: `https://scikit-optimize.github.io/stable/modules/generated/skopt.Optimizer.html` (cit. on p. 34).

[14] M.A. Mendez, A. Gosset, B. Scheid, M. Balabane, and J.M. Buchlin. «Dynamics of the Jet Wiping Process via Integral Models». In: (21 Nov 2020), p. 44 (cit. on pp. 42, 43, 50).

[15] Anne Gosset and Jean-Marie Buchlin. «Jet Wiping in Hot-Dip Galvanization». In: (2September 2019.) (cit. on p. 43).

[16] Christian Ruyer-Quil and Paul Manneville. «Improved modeling of flows down inclined planes». In: (2000) (cit. on p. 46).

[17] S. MUKHOPADHYAY, M. CHHAY, and C. RUYER-QUIL. «Modelling transitional falling liquid films». In: (1 September 2017) (cit. on p. 46).

[18] S. Beltaos. «Oblique impingement of plane turbulent jets». In: (1976) (cit. on p. 47).

[19] Adam Ritcey, Joseph Mcdermid, and Samir Ziada. «The Maximum Skin Friction and Flow Field of a Planar Impinging Gas Jet». In: (1 October 2017) (cit. on p. 47).

[20] «REINFORCEMENT LEARNING FOR ACTIVE FLOW CONTROL». In: *M. Desmet*. Von Karman Institute, Belgium, 2021 (cit. on pp. 57, 68).

[21] Fabio Pino, Lorenzo Schena, Jean Rabault, Alexander Kuhnle, and Miguel A. Mendez. «Comparative analysis of machine learning methods for active flow control». In: (February 2022), p. 47 (cit. on p. 65).

[22] *Kullback–Leibler divergence*. URL: `https://en.wikipedia.org/wiki/Kullback%5C%E2%5C%80%5C%93Leibler_divergence` (cit. on p. 77).

[23] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. «Trust Region Policy Optimization». In: (20 April 2017), p. 16 (cit. on p. 77).

[24] Richard S. Sutton. «Reinforcement Learning: An Introduction». In: *The MIT press* (2017) (cit. on p. 78).

[25] Diederik P. Kingma and Jimmy Lei Ba. «ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION». In: (2015), p. 15 (cit. on p. 79).