Learning Parametric Integer Programming via Inverse Optimization

Milan Dankovic

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Learning Parametric Integer Programming via Inverse Optimization

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

Milan Dankovic

October 11, 2021

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Delft University of Technology
Department of
Delft Center for Systems and Control (dcsc)

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical, Maritime and Materials Engineering (3mE) for acceptance a thesis entitled

Learning Parametric Integer Programming via Inverse Optimization

by

Milan Dankovic

in partial fulfillment of the requirements for the degree of

Master of Science Systems and Control

Dated: October 11, 2021

Supervisor(s):

dr. P. Mohajerin Esfahani

P. Zattoni Scroccaro

Reader(s):

dr. M. Mazo Espinosa

dr. B. Atasoy

Abstract

Inverse Learning is implemented in order to learn a control/decision policy (in the integer space) from an Expert Agent. The Learner Agent assumes that the Expert is acting minimizing an unknown cost function and tries to approximate it, through its own parametrized version of it. Learning can be performed in two different ways: offline (exploiting a training set containing Expert data) and online, in which the Learner Agent is directly controlling the system while learning the policy, exploiting corrective advice from the Expert. We propose three different learning algorithms and draw a comparison between them, as well as assess their performance against the Expert. We use our Agent for two different applications: control of a dynamical system (1) and classic ML classification/regression tasks (2). For application (1), our case study is the Heavy Shell Oil Fractionator system, with an MPC Expert Agent. For application (2), we train and test our Agent on several real data-sets available online, with tasks such as medical diagnosis, social media comment volume prediction, fault detection/isolation and multi-class classification.

Table of Contents

Acknowledgements

This Report is the conclusion of my Thesis Project and of my studies in the MSc "Systems & Control" at TU Delft. It has been a challenging and stimulating experience, that allowed me to develop new skills and deal with many interesting subjects and projects.

I would like to thank my supervisor dr. Peyman Mohajerin Esfahani, whose curiosity for research, positive attitude and never-ending energy have been a great drive for me to give my best during the project. Then, I would like to thank Pedro Zattoni Scroccaro, who provided great support and with whom I have spent many hours discussing details, results and ideas regarding the project. I would also like to thank the committee members Manuel Mazo Espinosa and Bilge Atasoy.

Additionally, I would like to thank my family who has always been very supportive throughout my whole academic career.

Delft, University of Technology — Milan Dankovic
October 11, 2021

Chapter 1

Introduction

The Control Engineering field has witnessed over the last decades an increasing interest towards numerical optimization, as the significant improvements in hardware and software tools have been allowing an ever larger exploitation of such techniques. A significant example is Model Predictive Control (MPC), which merges the classic Systems and Control framework with numerical optimization, in order to predict the system behaviour over a horizon and apply the optimal input (minimizing a certain cost function, which takes into account the desired penalization for each input and state/output variable) at each time step, respecting the imposed constraints. With ever more relevant improvements in computation times, MPC usage has become feasible even for systems with quick dynamics, making it a very solid and versatile type of controller ([Lee11], [Bem06]).

Another fruitful research direction in the field, which also exploits numerical optimization tools, is the use of learning techniques for control. For instance, Reinforcement Learning (RL) has become quite popular in several control applications ([KBP13]), as it is able to learn a control policy quite naturally, even in complex tasks in which a model is not available or not accurate. In fact, as it is inspired from animal learning, it functions model-free and updates its policy simply observing rewards and punishments connected to the control inputs it applies to the system. For that reason, with trial and error and exploration of several state/action possibilities, the Learner is able to improve iteratively its control policy.

On the other hand, it may be useful to learn a policy from an Expert Agent, which is already able to control a certain system. A possible approach in that direction is Imitation Learning ([LKYC16]), in which a Learner Agent achieves a direct mapping from a feature space (which can include for example the state of the dynamical system and the reference signal) to a control input space, from Expert Agent behaviour observations. It is thus a particular application of supervised learning, as the goal is to achieve a hypothesis function that fits the Expert data. This is however a limited approach, as it is only able to "imitate" the Expert behaviour in the observed situations, without having the possibility to solidly generalize the control action to new situations. A more involved approach is Inverse Learning

([BPS17],[BMPS20],[ESAHK17] and [AKE22]), in which the goal is to approximate the Expert control policy in a more solid way. In fact, the Learner Agent assumes that the Expert is acting on the system minimizing a cost function and tries to achieve a parametric approximation of that cost. In doing so, the Learner Agent does not simply mimic the actions of the Expert, but it is trying to capture the core of its behaviour, the way it is computing its actions, and approximate it. The Learner Agent can either be trained offline, using a previously achieved data-set containing observations of Expert behaviour, or directly online, while controlling the system and receiving corrective advice from the Expert.

The Inverse Learning approach can additionally be exploited in the different application of classification/regression. In fact, as the Learner Agent achieves a hypothesis function, which maps an input space to an output space (which in the previous Control Application are feature space and control action space, respectively), we can train it on a data-set containing a number of input attributes and one or more output attributes (which can be classes or variables to be predicted), for each data point. In this setting, the available output datapoints are treated as the Expert predictions and our Learner will try to achieve a prediction behaviour as close as possible to the Expert one (which in reality is simply the real output variable value). Exploiting the Inverse Learning approach, it will learn a cost function, which is to be minimized in order to achieve the desired output prediction.

We will introduce the Inverse Learning problem for Control in Chapter 2, present our proposed solution and the used Algorithms in Chapter 3, introduce the Control Application case study (high-dimensional Heavy Oil Fractionator MIMO System) in Chapter 4, discuss the achieved results with both Offline and Online Learning approaches respectively in Chapters 5 and 6, and in Chapter 7 analyze the results of our Learner Agent application in the additional classification/regression scenario, with several real data-sets available on the Internet. We draw a comparison between our results and those reported in literature, from authors who created the data-sets and/or used them as their case study ([WM90], [SWM93], [Sch87], [DE83], [BCG+04], [GK14], [CCA+09], [Mat20], [SSK15], [HJR78]).

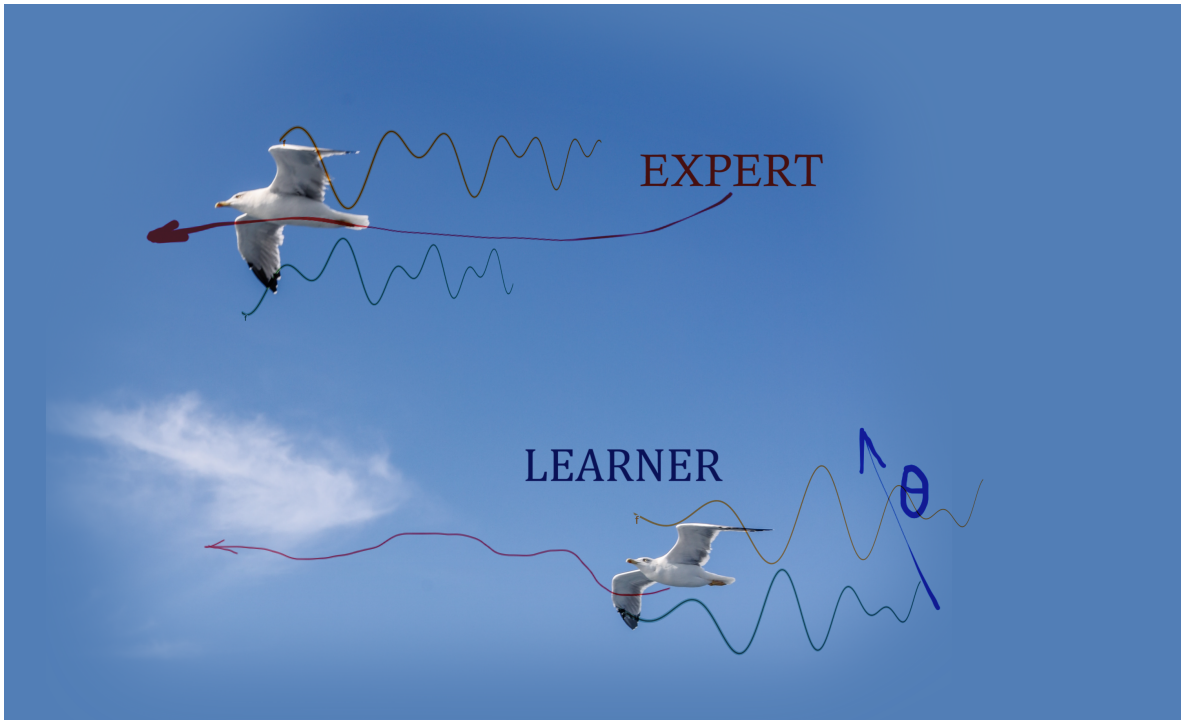


Figure 1-1: Learning from an Expert Agent

Chapter 2

Inverse Learning

2-1 Problem definition

Let us begin setting up a mathematical framework in the control problem setting, which is defined as follows. An Agent has the task of controlling a dynamical system (respecting certain requirements). In order to do so, at each time step t, it observes a feature vector $s_t \in \mathbb{S}$ and, exploiting that information, it computes the control input $u(s_t) \in \mathbb{U}$. The feature s_t defines the way that our Agent perceives the environment he is interacting with and contains the information that he can exploit for its learning and control tasks. For instance, the feature can be constituted by certain system quantities, such as dynamical state x_t or measured output y_t, and by quantities related to the control task, such as current reference value or the reference signal over a certain future horizon.

The core aspect of both Agents (Expert and Learner) consists in a mapping $h : \mathbb{S} \to \mathbb{U}$, from an input space $\mathbb{S} \subseteq \mathbb{R}^n$ to an output space $\mathbb{U} \subseteq \mathbb{R}^m$. The ultimate goal of the Learner Agent is to approximate as closely as possible the Expert control policy, which ideally would mean to learn the exact Expert mapping h^*. In order to do so, the Learner Agent assumes a certain structure of the mapping h^*. In fact, the assumption is that the Expert is controlling the system minimizing a certain unknown cost function $u \mapsto F^*(s, u)$:

$$h^*(s) \;=\; u^{et}(s) \;\coloneqq\; \arg\min_{u \in \mathbb{U}(s)} F^*(s, u). \tag{2-1}$$

On the other hand, in order to perform its learning task and approximate the unknown cost function $u \mapsto F^*(s, u)$, the Learner Agent must define a certain class of parametrized cost functions depending on the parameter θ, which is the decision variable of the learning process (and belongs to a certain admissible parameter space Θ):

$$\mathcal{F} \;=\; \{F_\theta(s, u) \;:\; \mathbb{S} \times \mathbb{U} \;\mapsto\; \mathbb{R} \;,\; \theta \in \Theta\}. \tag{2-2}$$

The Learner Agent computes its control action $u^{ln}(s)$ through its own parametrized hypothesis function, as follows:

$$h_\theta(s) \;=\; u^{ln}(s) \;\coloneqq\; \arg\min_{u\in\mathbb{U}(s)} F_\theta(s,u). \tag{2-3}$$

The Learner aims at achieving a parametrized hypothesis function that is the best possible approximation of the unknown Expert one, meaning that ideally, for each instance of the feature space $\mathbb{S}$, Expert and Learner control inputs coincide. In order to do so, the Learner must minimize, acting on the parameter θ, a loss function $l(u^{ln}(s), u^{et})$, which penalizes (through some metric) the disagreement between the two Agents control inputs. Assuming the availability of an Expert data-set $\{\hat{s}_t, \hat{u}_t\}_{t=1,...,T}$ containing T data points (with the $\hat{\cdot}$ notation we refer to the points available in the training data-set, achieved through observation of Expert behaviour), the learning process consists of the following optimization program:

$$\min_{\theta\in\Theta} \sum_{t=1}^{T} l(u^{ln}(\hat{s}_t)\ ,\ \hat{u}_t^{et}). \tag{2-4}$$

2-2 Inverse Learning Agent peculiarities

2-2-1 Advantages over the Expert Agent

The Expert Agent is controlling the system solving at each time step an optimization program that minimizes the cost function $F^*(s,u)$. In general, that cost function can be very costly from a computational perspective. Let us for example take into account the MPC controller ([Lee11], [Bem06]), which is the state of the art when considering the classic Control Theory Framework. Let us define the MPC problem for a reference tracking control scenario, with linear time-invariant system dynamics $x_{t+1} = Ax_t + Bu_t$ (where each t indicates a specific time-step). If the desired reference values for state and input are respectively x_t^{ref} and u_t^{ref}, let us define the MPC stage cost $c(x_t, u_t) = (x_t - x_t^{\text{ref}})'Q(x_t - x_t^{\text{ref}}) + (u_t - u_t^{\text{ref}})'R(u_t - u_t^{\text{ref}})$ (with weighting matrices Q and R) and the input sequence $\boldsymbol{u}_t = \{u_t, ..., u_{t+N-1}\}$ (with prediction/control horizon of length N). Then, the MPC cost can be defined as

$$W_N(x_t, \boldsymbol{u}_t) \;=\; \sum_{i=0}^{N-1} c(x_{t+i}, u_{t+1}) + V_f(x_{t+N}), \tag{2-5}$$

where $V_f(x_{t+N})$ is the final cost, at the end of the horizon. The complete problem formulation is

$$\begin{aligned}
\min_{\boldsymbol{u}_t} \quad & W_N(x_t, \boldsymbol{u}_t) && && \text{(2-6)}\\
\text{s.t.} \quad & x_{t+i+1} = Ax_{t+i} + Bu_{t+i}, && i = 0, ..., N-1 && \text{(2-7)}\\
& u_{t+i} \in \mathbb{U}(x_{t+i}), && i = 0, ..., N-1 && \text{(2-8)}\\
& x_{t+i} \in \mathbb{X}, && i = 0, ..., N. && \text{(2-9)}\\
& && && \text{(2-10)}
\end{aligned}$$

The MPC Agent is solving a complex program at each time step, optimizing the system evolution trajectories and the whole sequence of control inputs, over the horizon N, while respecting the desired constraints. However, ultimately, he only applies the first input of that sequence $\boldsymbol{u}_t$ and repeats the process at each time-step. His behaviour can be expressed in the more compact and general formulation of Eq. 2-1, as $u^{et} = \arg\min_{u \in \mathbb{U}(s)} F^*(s,u)$, which is the formulation we use in our framework. It is a very powerful tool, which ensures the desired performance even in challenging control scenarios. However, the optimization program can be computationally demanding, especially when dealing with long prediction horizons, high-dimensional MIMO systems and many constraints. Additionally, the computational complexity increases much more significantly if we are dealing with an integer control space, as large integer optimization programs are much more time-demanding (and in this project we are dealing with integer control). In fact, as briefly mentioned before, the computational complexity was one of the main drawbacks of this controller, and the one that prevented its use for quick-dynamics systems until hardware improvements managed to bridge the gap.

With the Inverse Learning Agent, we are able to approximate that complex Expert control policy in a much simpler parametrized cost function $F_\theta(s,u)$, which has a convenient structure. In fact, as we will see later, the optimization program that the Learner Agent must solve in order to compute its control input ($\arg\min_{u \in \mathbb{U}} F_\theta(s,u)$) is convex. That results in much faster computation times than the ones required by the MPC, for instance. Additionally, in the integer control scenario, it is possible to solve the program with a combinatorial approach. In fact, assuming that we have m inputs and each input can take c integer values, then c^m is the number of all the possible input vectors to the system. The one that solves the program $\arg\min_{u \in \mathbb{U}} F_\theta(s,u)$ can be found simply evaluating the cost for each possible input vector and selecting the one with the lowest cost. That approach results in even lower computation times, as evaluating the loss function for all the possible inputs and choosing the best one is much quicker than solving an integer optimization program. In fact, whatever the approach and solver we use (even advanced numerical tools such as Mosek and Gurobi), the optimization program for computing the desired control input must be initialized and run until convergence, which takes much more time than simply performing various evaluations of a function that additionally has a convenient structure (the specific parametrization is a design choice that we can act upon). That holds even if we have many inputs and a large set of admissible integer inputs. To give an idea of that computational time advantage, the MPC control input computation in our specific case study (in the changing reference tracking with integer control space scenario) takes on average 0.65 seconds, while the Learner Agent input computation time is 0.00056 seconds, which is 1171 times faster (with our specific hardware).

That means that we are able to capture the Expert control behaviour and encode it in a much simpler hypothesis function, which allows a comparable performance but with significantly faster computation times. We could even think to train the Learner Agent to behave as a very complex (and slow) Expert agent exclusively through numerical simulations (if we have an accurate and solid model) and apply it on the actual system with much faster computation times. More specifically, we could model the problem for a fast sampling time, which could not be sustained by the Expert in real time. But in numerical simulation, we can even allow a long computation for a control task in which the sampling time is much shorter. This way we could gather data from a complex Expert (which would not be applicable in reality) and then

approximate that control policy in a much simpler form and apply our fast Learner Agent on the actual system. Now, the Learner Agent could be able to keep up the quick sampling pace and at the same time apply (with a certain degree of approximation) the desired control policy.

2-2-2 Feature space role

The feature space plays a fundamental role for the Learner Agent, as that is the information he does use both for conducting the learning process and for computing its control inputs. The feature vector, for each time-step or data-point that is being considered, determines the way the Agent perceives the environment he is acting in. For instance, in a reference tracking control application, the relevant information for a control Agent may be, at each time step t, the dynamical state x_t of the system and the reference value x_{ref}. The feature vector, exploited at each time step t would then be

$$s_t \;=\; [x_t,\; x_t^{\text{ref}}]'.$$

His control policy is computed exploiting a parametrized cost function, over the feature space $\mathbb{S}$. For each feature instance $s \in \mathbb{S}$, he solves the program in Eq. 2-3 and maps that feature vector to his optimal control input. In other words, for each system configuration, seen through the lenses of the feature space, the Learner Agent follows a certain control behaviour (through the "arg min mapping" of the parametrized cost).

In broad terms, the Expert MPC Agent acts in the same way, minimizing however a more complex cost function and performing a complex optimization over possible system evolution trajectories at each step (see Eq. 2-5 and 2-6). On the other hand, the Learner Agent has a more direct feature-control input mapping, as it encodes in a very condensed way the control policy in its parameter θ. In the learning phase, it observes the Expert, approximating its behaviour in the parameter matrix while in the control phase, at each time instant, it simply receives a feature vector and knows how to act accordingly, through its parametrized policy. It does not predict constrained trajectories nor optimize a whole future sequence of inputs. It follows a more compact and simple approach that however has the same general structure the Expert has, as it is still solving the program $u^{ln}(s) \;=\; \arg\min_{u\in\mathbb{U}} F_\theta(s,u)$ for computing the control input. For that reason it achieves a more solid and general policy than, for instance, Imitation Learning does. However, in order for our Agent to properly generalize over the whole feature-space, some exploration is needed. Ideally, we need observations in each different region of system behaviour (in which we later intend to control the system). If we only learn on a limited data-set containing only observations in one portion of the feature-space, our parametrization will achieve a control policy suited for that region. It will generalize the learned control behaviour over the whole space (as we can always compute optimal inputs for new unforeseen feature configurations), but it might be imprecise in the unexplored regions, where the system might respond differently and different control strategies should be used. In the Appendix Section A-3 we provide a simple example that highlights this aspect.

It is important that the feature space used by the Learner Agent is sound to the specific application, in order for it to being able to capture the relevant information, either from

the training data-set (in the offline learning scenario) or from the Expert corrective advice (online scenario). In general, if our Agent uses the same information the Expert does, then the learning process is expected to be successful and lead to a control policy which yields a performance comparable to the Expert. It is even possible to use an enriched feature space for the Learner, in order to better exploit the available information. In fact, as it uses a data-driven approach, we can encode any possible feature space that we desire, from the available data. For instance, we could have $s_{\text{richer}} = [s; s^2; s^3]$ or $s_{\text{richer}} = [s; \sin(s); \cos(s)]$ or $s_{\text{richer}} = [s_i \cdot s_j]$, with $i, j = 1, ..., n$ (full quadratic extension, considering all the possible inter-feature products). We will actually use such an approach in the classification/regression applications.

2-3 Recent results and our project

The Inverse Learning approach was recently explored in [BPS17],[BMPS20],[ESAHK17] and [AKE22]. In [ESAHK17], authors introduce the Inverse Learning problem and create a theoretical framework, proposing tractable loss function and cost parametrizations (linear and quadratic). Additionally, they focus on probabilistic guarantees when noise is affecting the training data-set and on a distributionally robust approach that ensures a certain performance standard for the Agent. Successful numerical experiments are provided through offline optimization programs.

In [BPS17] and [BMPS20], two Inverse Learning iterative algorithms are explored, which can be implemented online and are compatible with (mix-)integer inputs. The authors use a linear cost parametrization for the Learner Agent and provide numerical results for such an approach, additionally focusing on convergence rates.

The most recent paper,[AKE22], was the starting point and played an important role in our work (we use a very similar mathematical framework and notation). The authors apply the Inverse Learning approach explicitly to a system control application. The problem is solved in a continuous control action space, using a quadratic cost parametrization, defining an efficient LMI problem reformulation and providing successful results on an actual simple physical set-up and for high-dimensional numerical simulations. Our paper starts from there and focuses mainly on three new aspects and challenges: integer control scenario, online learning/control and classification/regression tasks on real available data-sets. We mainly focus on the application of the algorithms, trying to extrapolate the main peculiarities and solidly assess their performance in a variety of tasks.

We start by defining our Inverse Learning Algorithms in the integer control scenario. We derive an optimization program that is able to solve the problem efficiently and find the best possible θ parametrization, exploiting a mathematical reformulation of the problem (it serves a similar purpose the LMI did in the continuous space, but now in the integer space). It finds the exact solution to the problem, but it requires solving a large optimization program, which could become intractable with complex problems and large data-sets. Additionally, we explore two iterative learning algorithms, (batch-)Subgradient Descent and Mirror Prox.These

algorithms are tested in numerical simulations on the Heavy Oil Fractionator case study. We first assess the three Agents learning/control performance in the offline scenario, highlight the differences of these approaches and draw a comparison with the Expert MPC Agent. At this point, with a deeper understanding of the problem and of the Learning Agents, we explore the online learning/control scenario (using batch Subgradient Descent and Exact Agents).

Finally, we exploit the Inverse Learning approach in a completely different scenario: the classic ML classification/regression task, performed on real data-sets found online. Some of the tasks are: Fault Detection/Isolation for an industrial system, predicting the number of comments for a Facebook post, diagnosing a sickness based on people's medical records and others. In doing so, we show the potential of this learning approach even outside the typical control scenario for which we initially designed it and show how versatile that algorithm is.

2-4 Notation

A brief overview on some notation we use. If we specifically refer to available data-points in the Expert data-sets (observations of Expert Agent controlling the system) the "hat" notation ($\hat{\cdot}$) is exploited. These available data-sets contain T instances, and we refer to each of these with the index $t \in \{1, ..., T\}$ (corresponding to each time-step observation). We refer with u^{et} to the Expert control input, achieved minimizing the Expert cost function $F^*(s, u)$. If we refer to an Expert control input data point (in the considered training set), at time-step t, the used notation is $\hat{u}_t^{et}$. If we refer generically to feature and control input, inside our framework, we simply denote them as s and u (they may also be in the available set, but we are considering them for a general reasoning). Regarding the Learner Agent, u^{ln} stands for the Learner optimal input, achieved through the program $\arg\min_{u \in \mathbf{U}(s)} F_\theta(s, u)$. Regarding the iterative learning processes, we refer to each iteration with k. Similarly, in the online learning/control scenario, k denotes each time-step, which also corresponds (during the online learning phase) to one learning iteration. The transpose matrix of A is denoted as A'.

Chapter 3

Our solution

3-1 Mathematical formulation

Let us begin from the problem and notation introduced in Section 2-1. The goal of the Learner Agent is to achieve a parametrized cost function $F_\theta(s,u)$ which is the closest possible approximation to the Expert cost $F^*(s,u)$. In order to do so, a loss function $l(u^{ln}(s), u^{et})$ (which penalizes the two Agents' control inputs disagreement) is to be minimized. We follow the mathematical formulation proposed in [AKE22]. Let us define a **quadratic** parametrization for the Learner cost function. The cost functions class considered by the Learner is:

$$\mathcal{F} = \left\{ F_\theta(s,u) = \begin{bmatrix} s \\ u \end{bmatrix}' \theta \begin{bmatrix} s \\ u \end{bmatrix} \mid \theta \in \Theta \right\}. \tag{3-1}$$

We can express the parameter matrix θ with its block components

$$\theta = \begin{bmatrix} \theta_{ss} & \theta_{su} \\ \theta_{us} & \theta_{uu} \end{bmatrix}. \tag{3-2}$$

As the Learner Agent control policy is achieved minimizing such cost over the u variable (Eq. (2-3)), the block θ_{ss} does not play a relevant role and can be set to zero. Additionally, we define the parameter matrix to be symmetric ($\theta_{us} = \theta'_{su}$), resulting in:

$$\theta = \begin{bmatrix} 0 & \theta_{su} \\ \theta'_{su} & \theta_{uu} \end{bmatrix}. \tag{3-3}$$

In order for the optimization program in the learner hypoyhesis function (Eq. (2-3)) to be convex, we must impose that the block θ_{uu} is positive definite ($\theta_{uu} > 0$). Furthermore, we impose $\theta_{uu} \geq I$ (all eigenvalues larger or equal than one), as that has simply the effect of scaling the parameter matrix and does not affect the learning process. In doing so, we avoid

the risk of θ entries converging towards zero during the learning process, making it numerically more reliable.

Regarding the **loss function**, which is to be minimized, we exploit the sub-optimality loss:

$$l^{sub}(s, u^{et}, u^{ln}) \;\coloneqq\; F_\theta(s, u^{et}) \;-\; F_\theta(s, u^{ln}) \tag{3-4}$$

$$l^{sub}(s, u^{et}, u) \;\coloneqq\; F_\theta(s, u^{et}) \;-\; \min_{u \in \mathbf{U}(s)} F_\theta(s, u). \tag{3-5}$$

It is a peculiar loss function and it serves well our task. In fact, it does not assume knowledge of the true cost F^*, as it penalizes the difference between the two Agents' control inputs through the parametrized cost F_θ. Such a loss function is zero (for a certain feature instance s) when these inputs coincide, meaning that the minima of the Expert true cost and of the Learner parametrized variant coincide. Regarding the notation, in Eq. 3-4, the loss is expressed in a more compact form, with u^{ln} being the optimal Learner's input (so the inner minimization program is implicit). On the other hand, in Eq. 3-5, we have the general loss formulation, explicitly expressing the inner minimization program. Additionally, such a loss is a convex function w.r.t. θ. In fact, as $min_{u \in \mathbf{U}} F_\theta(s, u) \;=\; -max_{u \in \mathbf{U}} \{-F_\theta(s, u)\}$, the loss function is the result of a maximization over a piece-wise linear function in θ (see Eq. 3-6).

$$l^{sub}(s, u^{et}, u) \;= \max_{u \in \mathbf{U}(s)} \{\; F_\theta(s, u^{et}) \;-\; F_\theta(s, u) \;\}. \tag{3-6}$$

Ultimately, given an Expert data-set $\{\hat{s}_t, \hat{u}_t\}_{t=1,\dots,T}$ containing T data points, the Inverse Learning problem consists in:

$$\min_{\theta \in \Theta} \sum_{t=1}^{T} l^{sub}(\hat{s}, \hat{u}^{et}, u^{ln}) \;= \min_{\theta \in \Theta} \sum_{t=1}^{T} \Big\{ \max_{u \in \mathbf{U}(s)} \{\; F_\theta(\hat{s}, \hat{u}^{et}) \;-\; F_\theta(\hat{s}, u) \;\} \Big\}. \tag{3-7}$$

3-2 Exact solution

It is possible to solve the Inverse Learning problem in Eq. 3-7 (for the Integer control scenario) with one optimization program and achieve the exact solution to the problem, that is, the best possible parameter θ achievable with the Inverse Learning approach. In fact, as shown in [L.V13], with a support variable p, we can reformulate the problem as follows:

$$\min_{\theta, p_t} \quad \sum_{t=1}^{T} p_t \tag{3-8a}$$

$$\text{s.t.} \quad l^{sub}(\hat{s}_t, \hat{u}_t^{et}, u_t) \le p_t \quad \forall u_t \in \mathbf{U}, \quad \forall t \in [1, \dots, T] \tag{3-8b}$$

$$\theta_{uu} \ge I \tag{3-8c}$$

All the possible loss function evaluations ($\forall u_t \in \mathbf{U}, \quad \forall t \in [1, \dots, T]$) are encoded in the constraints, with the support variable p_t being pushed against them. This way, the inner

maximization in Eq. 3-7 is directly encoded in this problem formulation. The program optimizes for θ, finding the best possible cost parametrization for the Learner Agent. This approach is computationally very demanding, as it considers all the possible control input combinations for all time steps, and the problem can become very large, especially with high-dimensional problems and wide data-sets.

3-3 Iterative approach

As the Exact program can become computationally expensive or even intractable, we will exploit two iterative algorithms for the Inverse Learning Agent: Subgradient Descent and Mirror Prox. These algorithms refine the parameter θ estimate at each iteration, exploiting the previous estimate and a learning rate, and do not solve the whole problem at once, as happens in the Exact Learning Agent. They achieve a sub-optimal solution w.r.t. the Exact one (which in fact achieves the best possible minimizer of the sub-optimality loss). In the "Numerical Results" Chapters, we assess their performances (which are comparable), as well as their learning processes. The Mirror Prox algorithm, proposed by Nemirovski ([Nem04]) achieves a convergence rate $O(1/k)$) (with k being the number of learning iterations), faster than Subgradient Descent (whose convergence rate is $O(1/\sqrt{k})$ and it is applicable to smooth saddle-point problems. For example, such a type of problem can be expressed with the generic structure

$$\min_{x\in\mathbf{X}} \max_{y\in\mathbf{Y}} \; xzy, \tag{3-9}$$

where $x \in \mathbf{X}$ and $y \in \mathbf{Y}$ are optimization variables and $z \in \mathbf{Z}$ is a constant. It is a saddle-point problem as one optimization variable (x) is being minimized, while the other (y) is being maximised. Furthermore, it is smooth, as the objective function xzy is linear in both the optimization variables. In our Inverse Learning scenario, it is possible to reformulate the problem in the structure of Eq. 3-9. We derive the reformulation, explain the algorithm and provide a graphical scheme of how it works in the Appendix (A-1 and A-2). On the other hand, in the next section we define the Subgradient Learning algorithm.

3-3-1 Subgradient descent

This approach exploits the Subgradient w.r.t. θ at each learning iteration, in order to improve the parameter approximation. As the sub-optimality loss is linear in θ, the Subgradient that takes into account one data-point (s, u^{et}) is simply computed as

$$\nabla l^{sub}(s, u^{et}, u^{ln}) \;=\; \begin{bmatrix} s \\ u^{et} \end{bmatrix} \begin{bmatrix} s \\ u^{et} \end{bmatrix}' \;-\; \begin{bmatrix} s \\ u^{ln} \end{bmatrix} \begin{bmatrix} s \\ u^{ln} \end{bmatrix}', \tag{3-10}$$

where u^{ln} is the solution of the program in Eq. 3-6. The Subgradient Descent update is

$$\theta^{+} \;=\; \Pi(\,\theta \;-\; \eta \nabla l^{sub}(s, u^{et}, u^{ln})), \tag{3-11}$$

with η learning rate and Π projection operator. The projector operator Π is needed to ensure that the previously discussed constraint $\theta_{uu} \geq I$ is respected. In order to do so, we perform the eigenvalue projection on θ_{uu}, which sets to one all the eigenvalues lower than one:

$$\Pi(\theta_{uu}) \;=\; \sum_{i=1}^{m+n} max\{\lambda_i,\ 1\} v_i v_i', \tag{3-12}$$

where v_i and λ_i are the corresponding eigenvectors and eigenvalues of θ_{uu}. We use the dacaying learning rate $\eta = \eta_0/\sqrt{k}$ (with k the current learning iteration) in order to ensure convergence. When we will report the used learning rate for the experiments in the "Numerical Results" Sections, we will refer to the initial value η_0, which then decreases during the experiment.

It is possible to compute the Subgradient using multiple data-points (and even the whole available data-set), summing the Subgradient value achieved at each exploited data point. For instance, exploiting all the available information at each step and solving the problem in Eq. 3-7, we get

$$\nabla l^{sub,T} \;=\; \sum_{t=1}^{T} \begin{bmatrix} s_t \\ u_t^{et} \end{bmatrix} \begin{bmatrix} s_t \\ u_t^{et} \end{bmatrix}' \;-\; \begin{bmatrix} s_t \\ u_t^{ln} \end{bmatrix} \begin{bmatrix} s_t \\ u_t^{ln} \end{bmatrix}'. \tag{3-13}$$

Here, it should be noted that u_t^{ln} depends implicitly on θ (as it is achieved minimizing the parametrized cost). It is possible to have a Batch Subgradient Descent algorithm, selecting a subset (batch) of available data-points for each iterative update. The Subgradient is then computed performing a summation similar to the one shown in Eq.3-13, but only over the data-points of the batch.

The Subgradient Descent approach can successfully be exploited for online learning/control (Learner is controlling the system, while learning and improving the parameter estimate, exploiting Expert corrective advice). In that case, at each time step k, the Subgradient is computed using last available b data points $\{s_i, u_i^{et}\}$, with $i = k-b, ..., k$ (that aspect is further explored in Chapter 6). Based on the available computational time for the real time Learner computation and on the memory size of the controller, we can design a certain batch size b for such an application.

3-4 Numerical tools and used hardware

The machine we use for all the numerical experiments has the following properties: processor AMD A10-9600P, clock speed 2.40 GHz, RAM 16 GB, OS Windows10 x64. All the programs and experiments are coded and run in MATLAB R2019b. We additionally use the Yalmip toolbox and interface for some optimization programs ([Lof04]), with the availability of Gurobi and MOSEK solvers.

Chapter 4

Case study: Heavy Shell Oil Fractionator

4-1 System overview

We test our Inverse Learning Agent on the Heavy Shell Oil Fractionator system, with an MPC controller as Expert Agent. We refer to [Mac02] for a more detailed system description, the dynamical model and more insight on the control task and on the optimal MPC controller design (we follow the general problem setting and some design choices from that source, such as for instance the sampling time and the way constraints are being dealt with).

To begin with, in large industrial plants, there may be the need to fractionate one large source of heat (which takes the form of a gaseous feed), divide it and control it, in order to ultimately convey it in a controlled and measured way to several parts of the plant, where that heat/gas is required. For that reason, each fractionator in the plant can either be connected to smaller intermediate ones or directly to plant chambers where heat or gas has to be conveyed. The fractionator system we are dealing with consists of a main tank, at the bottom of which heath is being introduced. It is connected (at the top and at the side) to two smaller tanks, which can draw heat from it. In our control task, we are considering three measured outputs and three control inputs. Two analyzers measure the Top and Side End Point Compositions (y_1 and y_2 respectively), which indicate the property of the gaseous feed that is entering the two smaller tanks. Additionally, we measure the temperature at the bottom of the main tank (y_3). Regarding the control inputs, we can act on the Top and Side Draw (u_1 and u_2),which determine the amount of heat being transferred to each of the smaller tanks, and on the Bottom Reflux Duty (u_3), which determines how much heat is being recirculated back to the bottom of the main tank. We follow the original design choice of making the Bottom Reflux Duty an additional measured output (y_4), as for the control application it is regarded as a performance output (a quantity of interest that is to be controlled). As it coincides with the third input, we have that $y_4(k) = u_3(k-1)$. For a visualization of the system, see Fig. 4-2

Figure 4-1: Heavy Oil Fractionator
Image: https://www.ecolab.com/solutions/primary-fractionator

in which inputs are highlighted with a blue box and outputs with a red one.

As described in [Mac02], a continuous LTI state-space model is achieved from input-output transfer functions and then the model is discretized with a 4 minutes sampling time. A 32-dimensional MIMO system is achieved, with four outputs and three control inputs. The control task is reference tracking (we will explore both Single Reference and Changing Reference control tasks) while respecting the constraints. In fact, each variable must be inside the range $[-0.5, 0.5]$ and, additionally, there is the input move constraint $|\Delta u_i| \leq 0.05 \cdot T_s$ for $i = 1, 2, 3$, meaning that each control input cannot change more than ± 0.2 at each time step. We complicate the task even more, as we are working in an integer control scenario. Each input can take a value in an integer set $u_i \in \{-1,\ 0,\ +1\}$ for $i = 1, 2, 3$. The actual applied control value is $\alpha \cdot u$, where α plays the role of a transducer constant and is set to the value 0.2.

Regarding the constraints, following the original Maciejowski's design, we encode them in the MPC program only for the later steps of the horizon, leaving the first 5 steps unconstrained (that value was hand-picked to ensure the best compromise, after several numerical experiments). This is done mainly to always ensure feasibility of the MPC program (as in certain specific configurations, the program could incur in infeasibility, which is undesirable both in a real application and in numerical simulation) and has the beneficial side effect of making the program slightly less computationally demanding and the control action more reactive. That causes occasional minor constraint breaking events, which are not problematic for the control task.

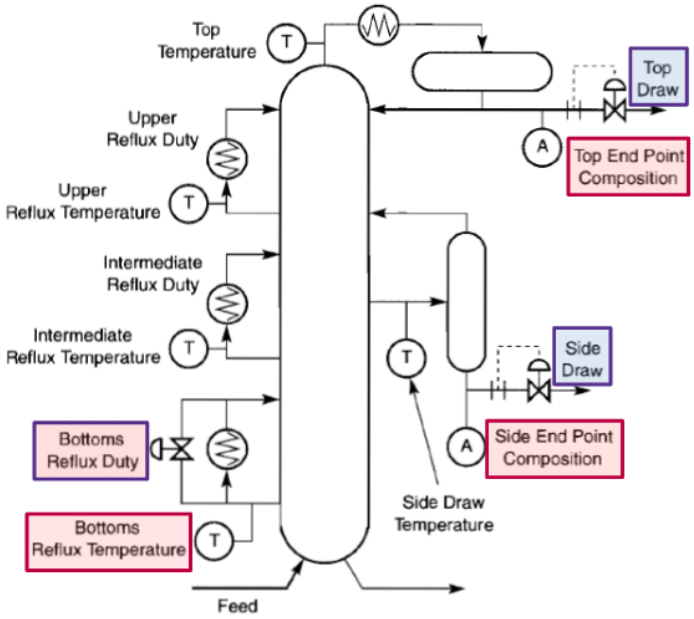


Figure 4-2: Heavy Oil Fractionator Scheme
image taken from [Mac02]

4-2 Single reference control scenario

In this control task (which is the same dealt with in [Mac02]), we assume that the fractionator is one of the largest ones in the plant and its objective is to fractionate the largest possible amount of heat. That is achieved minimizing the Bottom Reflux Duty ($y_4(k) = u_3(k-1)$), meaning that the lowest possible amount of heat is being recirculated back at the bottom of the tank. Additionally, Top and Side End Compositions (y_1 and y_2) are to be controlled as precisely as possible to zero (with more regard for y_1), while the Bottom Temperature (y_3) must simply respect the constraints. As we are in the integer control scenario, our inputs are in the set $\{-0.2,\ 0,\ +0.2\}$, so minimizing the Bottom reflux Duty means setting the reference for it at -0.2. Ultimately, the control requirement is to track the reference $y_{\text{ref}} = [0,\ 0,\ \sim,\ -0.2]'$ (the symbol $\sim$ indicates that there is no specific reference for y_3), while respecting the constraints.

We design a full state feedback MPC controller for tracking the output reference y_{ref}, with horizon $N = 15$ and the following weighting matrices:

$$\mathrm{Q} = \begin{bmatrix} 20 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathrm{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4-1}$$

In this scenario, we are using the classic causal MPC formulation, as it is being referred to in [Bem06]: at each time step k, the controller receives the current reference value y_{ref} and

uses that value for his computations, over the whole future horizon. Maintaining MPC cost and problem formulation similar to those in Eq. 2-5 and 2-6, we have that, at time-step t, the cost for output reference tracking is defined as:

$$W_N(y_t, \boldsymbol{u}_t) = \sum_{i=0}^{N-1} (y_{t+i} - \underline{y_t^{\text{ref}}})' Q (y_{t+i} - \underline{y_t^{\text{ref}}}) + (u_{t+i} - \underline{u_t^{\text{ref}}})' R (u_{t+i} - \underline{u_t^{\text{ref}}}) + V_f(y_{t+N}). \quad (4\text{-}2)$$

Observing the underlined terms in Eq. 4-2 it is possible to notice how the considered reference values (y_t^{ref} and u_t^{ref}) over the horizon (so for time steps $t, ..., t+N$) are those relative to the current time step t. It means that the MPC Agent does not have a preview on the future reference values (and for that reason is causal). In the specific control task, that does not make any difference, as the reference is constant (we will add preview of future references in the Changing Reference scenario).

On the other hand, the Learner Agent has the same information the Expert has, encoded in the feature vector $s(k) = [x(k),\ y_{\text{ref}}(k),\ 1]$. We also add a different Gaussian noise sequence on each of the 32 state variables in all the numerical simulations, in order to further complicate the task and introduce the model uncertainty that is present in the real application. Each noise sequence is achieved sampling randomly from $0.02 \cdot \mathcal{N} \sim (0, 1)$, where $\mathcal{N} \sim (0, 1)$ is a Gaussian distribution with zero mean and standard deviation (std.) equal to one.

4-3 Changing references control scenario

In this control task, we have a different reference sequence for each of the outputs y_1, y_2 and y_4. Each sequence is piece-wise constant, changing values every N (MPC horizon length) steps. The values present in each reference signal are taken from the set $\{-0.2,\ -0.1,\ 0,\ 0.1,\ 0.2\}$. So in total we have $5^3 = 125$ possible reference values (5 values taken by three variables), each one lasting 15 time-steps, for simulations lasting in total $125 \cdot 15 = 1875$ steps. We experiment both with and without gaussian dynamics noise. It is a quite complex control task, as the three outputs must track reference signals that are changing independently and rapidly. For that reason, we improve our Agents.

We designed an MPC Agent which has preview over the next N reference values. That means that he has a preview over the reference changes and is able to anticipate them, resulting in a much more reactive tracking behaviour ([Bem06]). Maintaining a consistent MPC cost formulation to Eq. 4-2, at each time-step t we get:

$$W_N(y_t, \boldsymbol{u}_t) = \sum_{i=0}^{N-1} (y_{t+i} - \underline{y_{t+i}^{\text{ref}}})' Q (y_{t+i} - \underline{y_{t+i}^{\text{ref}}}) + (u_{t+i} - \underline{u_{t+i}^{\text{ref}}})' R (u_{t+i} - \underline{u_{t+i}^{\text{ref}}}) + V_f(y_{t+N}). \quad (4\text{-}3)$$

This formulation is, strictly speaking, non-causal, as the Agent is computing its control inputs exploiting information from future time instances (see underlined reference values in Eq.4-3).

However, in such a task, it is completely reasonable to assume that the next N reference values are available (and in general that is the way MPC is implemented for Reference Tracking problems). The prediction/control horizon is $N = 15$ and the weighting matrices are:

$$\mathrm{Q} = \begin{bmatrix} 20 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix} \quad \mathrm{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4-4}$$

Regarding the Learner Agent, we experimented with four different Learner types, with increasing richness of feature space.

$$\begin{aligned} \text{Lear1} \quad &\rightarrow \quad s(k) = [x(k);\ y_{\text{ref}}(k);\ 1] \\ \text{Lear2} \quad &\rightarrow \quad s(k) = [x(k);\ y_{\text{ref}}(k);\ y_{\text{ref}}(k+N);\ 1] \\ \text{Lear3} \quad &\rightarrow \quad s(k) = [x(k);\ y_{\text{ref}}(k);\ y_{\text{ref}}(k+N);\ \Delta t;\ 1] \\ \text{Lear4} \quad &\rightarrow \quad s(k) = [x(k);\ y_{\text{ref}}(k+i);\ 1] \quad with \quad i = 0, ..., N \end{aligned}$$

"Lear1" is the type of learner that we used in the Single Reference scenario. Feature of "Lear2" also contains the next reference value, after N steps. "Lear3" takes additionally into account Δt, which is the time till the next reference change (for instance $\Delta t \in 1, ..., (N-1)$, when the reference changes every N steps). Finally, "Lear4" contains the whole reference sequence over the next N steps, so contains the same information that the MPC Expert has.

Learners 1 to 3 do not manage to learn an effective policy at all, failing to capture the Expert control behaviour. Sub-optimality loss and θ values converge, but the policy does not track the reference and actually very often the control inputs are kept at zero-value. On the other hand, Learner 4 manages to learn a meaningful and effective control policy, achieving a performance comparable to the Expert and exhibiting successfully the anticipative action behaviour.

It is then clear the important role played in the learning process by the feature space. In fact, it has to match the characteristics of the training data and be structured in a way to appropriately encode and capture the desired information. For both the control scenarios in our case study, the Learner Agents proved to be successful when taking into account the information that is also available to the Expert, which is a reasonable result. We test the Learner Agent on this case study, both in offline and online learning scenarios, assess its performance, compare the learned control behaviour with the MPC Expert one and analyze the learning process itself, drawing a comparison between the three proposed learning algorithms. All the results are provided in the following Chapters.

Chapter 5

Numerical results - Offline Learning

In the offline learning scenario, our Agent learns the control policy from an Expert data-set $\{\hat{s}_i, \hat{u}_i^{et}\}_{i=1,...,T}$. After the learning process is completed (using one of the three proposed approaches in Chapter 3) and the best parameter matrix θ for the available data-set is achieved, we test the Learner Agent in the desired control task (either Single Reference or Changing Reference Tracking, as explained in Chapter 4). In order to have a solid and consistent testing and assessment routine for the several possible scenarios that we explore (various learning algorithms for two different control tasks), we will proceed with In-Sample and Out-of-Sample testing.

In the **In-Sample** testing, we compute the Learner Agent control inputs on the data-points present in the training set (so in-sample, as we are exactly using samples exploited for the learning process). For each data-point $\hat{s}_i$ $(i = 1, ..., T)$, our Agent minimizes its learned cost $F_\theta(s, u)$ and computes the control input that he would apply to the system. At the end of the process, we gather the T control inputs that the Learner would have applied (instead of the Expert Agent) and directly compare them with the actual inputs that were applied by the Expert. So we can observe the control input disagreement of the two Agents. We compute it for each data-point $\hat{s}_i$ as $\|u^{ln}(\hat{s}_i) - u^{et}(\hat{s}_i)\|_1$ (for $i = 1, ..., T$) and compute the average disagreement value as well. From that, we additionally compute the accuracy of the Learner Agent, that is, the percentage of the instances in which he agrees with the Expert.

In the **Out-of-Sample** test, on the other hand, the Learner Agent (after training is completed on the Expert data-set) is directly controlling the system, generating a new trajectory and new data-samples (hence, out of sample). In fact, as the learned policy is an approximation of the Expert one, it will lead to a slightly different control behaviour and generate a different system evolution trajectory. Furthermore, in order to more extensively test our Agent, we apply a different dynamics noise sequence than the one used in the Expert data-set and, in the changing reference scenario, reshuffle multiple times the reference signal, in order to make sure that we are really in an Out-of-Sample situation and that the learned policy is solid. We compute the Tracking Error ($\sum_{i=1}^{T} |y_i - y_i^{ref}| \; \forall \; i$) and the QR stage cost ($y'Qy \; + \; u'Ru$,

where Q and R are the MPC weighting matrices). Subsequently, we achieve a histogram of all the QR cost instances and compute mean and standard deviation (std.). From there we can directly compare Leaner and Expert histograms, in order to observe the similarity of the two policies and especially to what extent the Learner is able to capture the noise rejection properties of the Expert.

5-1 Oil Fractionator: single reference

For this scenario, we use 1000 Expert data points for the training phase (taken from the complete Expert data-set made of 30000 data-points). Then, we test the learned Θ parametrization both In and Out-of-Sample. For the latter we conduct multiple long tests in order to achieve solid results. More in detail, we conduct five control tests, lasting 30000 time steps each. Every test has a different random (gaussian) dynamics noise sequence acting on the system. This way, we have a double benefit. Firstly, with long simulations, the stochasticity is properly taken into account and the histograms achieved for the QR Cost provide an accurate indication of the Agents' noise rejection behavior and allow for a solid comparison. Secondly, performing multiple tests, we can actually verify the performance of the learned Θ with several noise sequences (various long simulations ensure that we test system-controller-noise in remarkably numerous possible configurations) and get a solid confirmation that it ensures good performance for the desired control task.

We test the three different learning approaches explained in Chapter 3: Exact solution (achieved with one large optimization program), Subgradient Descent (with $\eta_0 = 0.005$ and 10000 iterations) and Mirror Prox ($\eta_a = 0.002$, $\eta_b = 0.01$ and 5000 iterations).

First of all, the performance of the learned control policy is very solid and comparable to the Expert one. Let us now analyze the details, drawing a comparison between Exact-Subgradient-MP solutions along the way. From the **In-Sample test**, we can observe that the learned control policy is slightly different from the Expert one, as they do sometimes disagree. In the following table it is possible to observe how many times Expert and Learner Agent agree on the control input (under the row "agrees") and the percentage of agreement in the whole In-Sample test (under the row "accuracy").

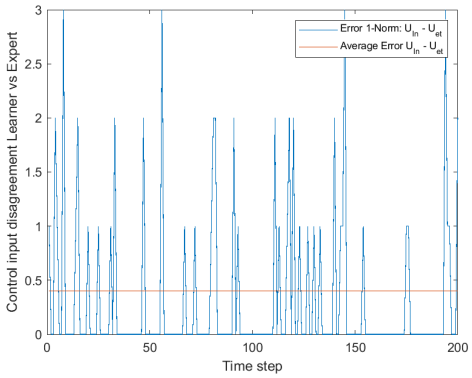
	Exact	Subgrad.	MP
agrees	22708	19947	22268
accuracy	75.69%	66.49%	74.23%

(5-1)

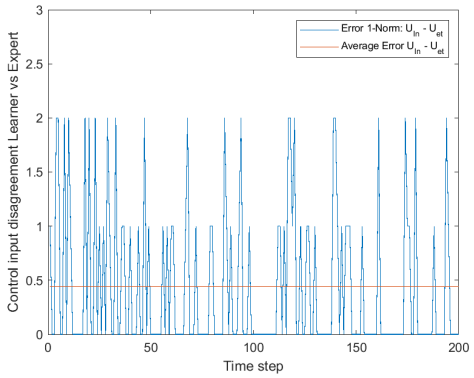
(5-2)

The Exact solution is the one with the highest accuracy, closely followed by the MP Agent, and then by the Subgradient one, which disagrees more often with the Expert. This is an expected behaviour, as the two iterative algorithms provide a sub-optimal solution wrt. the Exact one, meaning that they differ more from the target Expert policy. In Fig. 5-1, it is possible to visualize the control input disagreement ($\|u_{ln} - u_{et}\|_1$). The Exact Learner Agent disagrees less often with the Expert than Subgradient and MP Agents do, but when that happens the disagreement is generally larger. In order to have a clear comparison, we compute

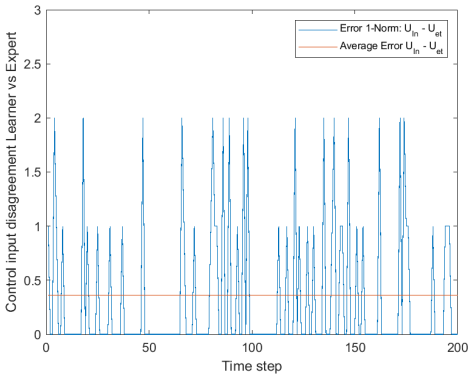
the average disagreement, achieving 0.4015 for the Exact solution, 0.4384 for the Subgradient one and 0.3609 for MP. Interestingly, the average MP disagreement value is lower than the Exact one, even if is slightly less accurate (as seen in Table 5-1). That happens because the disagreement, when happens, has a lower amplitude, never exceeding the value 2 (see Fig. 5-1). So, over the whole experiment, it achieves on average a closer behaviour to the Expert.



(a) Exact θ Agent



(b) Subgradient Agent



(c) MP Agent

Figure 5-1: Measure of disagreement $\|u_{ln} - u_{et}\|_1$

That happens because the Exact program pushes the numerics of the parameter at the maximum, in order to fit the training data at the best. And in doing so, it learns a policy which is closer to the Expert one (higher accuracy and better Out-of-Sample performance, as we will soon show). However, the parameter matrix is numerically uneven, containing entries with quite different orders of magnitude. That aspect, in certain In-Sample data-points, can lead to a larger disagreement.

Let us now analyze performance in multiple Out-of-Sample tests. Each column corresponds to one of the multiple experiments, and contains the tracking errors of each system output (there are four of them). The tracking errors are:

$$TrackErr_{Exact} = 10^3 \cdot \begin{array}{|ccccc|}\hline 1 & 2 & 3 & 4 & 5 \\ \hline 1.1665 & 1.1659 & 1.1567 & 1.1646 & 1.1555 \\ 1.3611 & 1.3620 & 1.3610 & 1.3548 & 1.3682 \\ 4.9132 & 4.9169 & 4.9175 & 4.9050 & 4.9246 \\ 2.3163 & 2.3159 & 2.3066 & 2.3085 & 2.3057 \\ \hline \end{array} \tag{5-3}$$

$$TrackErr_{Subgr} = 10^3 \cdot \begin{array}{|ccccc|}\hline 1 & 2 & 3 & 4 & 5 \\ \hline 1.1872 & 1.1874 & 1.1845 & 1.1849 & 1.1755 \\ 1.5712 & 1.5713 & 1.5692 & 1.5669 & 1.5837 \\ 4.6398 & 4.6398 & 4.6490 & 4.6610 & 4.6487 \\ 2.2774 & 2.2775 & 2.2720 & 2.2745 & 2.2728 \\ \hline \end{array} \tag{5-4}$$

$$TrackErr_{MP} = 10^3 \cdot \begin{array}{|ccccc|}\hline 1 & 2 & 3 & 4 & 5 \\ \hline 1.1592 & 1.1765 & 1.1644 & 1.1705 & 1.1604 \\ 1.4291 & 1.4152 & 1.4130 & 1.4117 & 1.4292 \\ 4.6817 & 4.6787 & 4.6888 & 4.7069 & 4.6765 \\ 2.2683 & 2.2600 & 2.2591 & 2.2600 & 2.2561 \\ \hline \end{array} \tag{5-5}$$

$$TrackErr_{Expert} = 10^3 \cdot \begin{array}{|c|}\hline 1.1378 \\ 1.4184 \\ 4.7761 \\ 2.2751 \\ \hline \end{array} \tag{5-6}$$

(5-7)

From the Tracking Errors, we can observe that the performance of the Learner Agents is comparable to the Expert Agent. As expected, the Exact Agent is the best one, over the multiple experiments, closely followed by the MP one and then by the Subgradient one, which is a bit worse. Anyhow, their performance is comparable, as the numerical differences in the Tracking Errors are not relevant considering that each experiment lasts 30000 time steps and that the weighting matrix Q contains large values (Q=diag(20,10,0,1)). It is interesting to notice, however, that the the Exact Agent achieves a lower tracking error for y_2 than the Expert (but a higher one for y_1), in all the experiments. That means that, our program managed very effectively to extrapolate from the training data some information underlying the behaviour of the second output, and in deriving its policy (which differs inevitably from the Expert) it exploits that knowledge. Similarly, MP Agent achieves a lower tracking error for y_2 than the Expert in three experiments and always for y_4 (which we recall is equivalent to u_3).

Let us now analyze the QR Cost histograms. In order to achieve them, we compute the QR cost $(y_t - y_t^{\text{ref}})'Q(y_t - y_t^{\text{ref}}) + (u_t - u_t^{\text{ref}})'R(u_t - u_t^{\text{ref}})$ at each time-step t of the Out-of-Sample test and store it as an entry of a vector. In the end, it is possible to plot a histogram of all the cost occurrences present in the vector, observe how the QR cost values are distributed and compute information such as mean and standard deviation (std). These histograms can be observed in Fig. 5-3, plotted against the MPC Expert one (achieved during Expert control experiments). In these plots, the mean is represented by a colored ball and std by a horizontal line, whose length encodes the numerical value. For the Learner Agents, we show the histograms in the worst case scenario, the second experiment (out of five), in which the

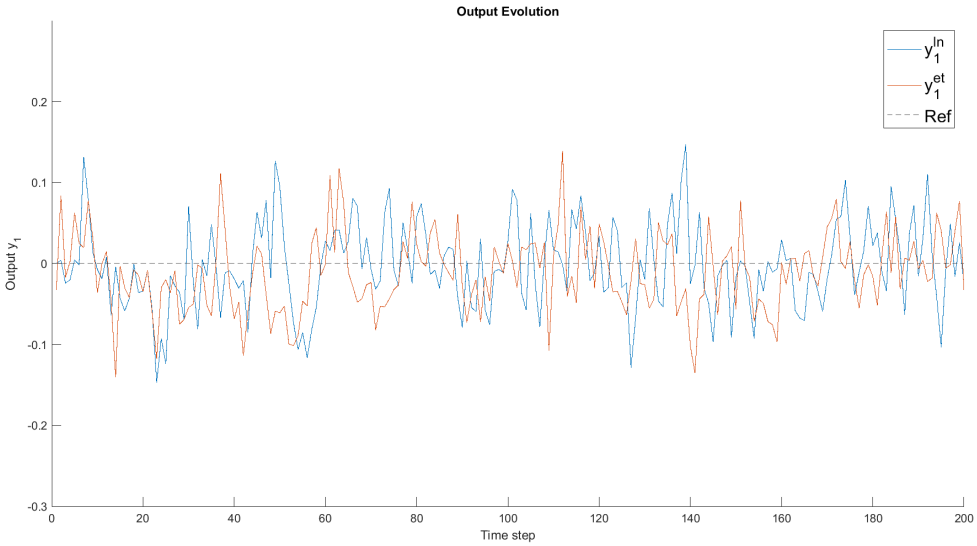


Figure 5-2: Subgradient Agent vs MPC (y_1)

QR cost is on average the highest (due to the specific numerical aspects of the noise sequence we use), as can be seen in 5-11. The large duration of the experiment ensures a reliable histogram, with stochasticity being properly held into account.

QR Exact	1	2	3	4	5
mean	1.1367	0.1369	0.1359	0.1363	0.1363
std	0.0859	0.0858	0.0848	0.0856	0.0856

(5-8)

QR Sub	1	2	3	4	5
mean	0.1374	0.1378	0.1370	0.1373	0.1374
std	0.0938	0.0938	0.0932	0.0933	0.0937

(5-9)

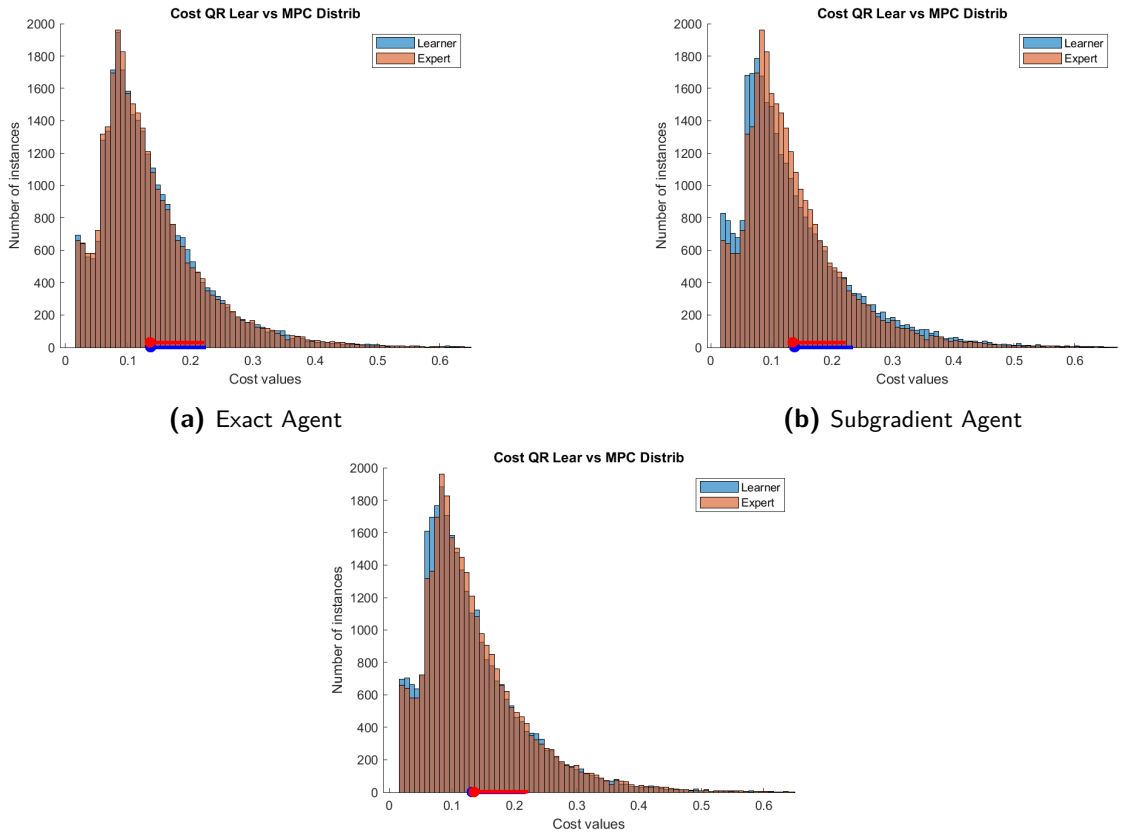
QR MP	1	2	3	4	5
mean	0.1318	0.1321	0.1312	0.1317	0.1317
std	0.0851	0.0840	0.0838	0.0843	0.0839

(5-10)

QR MPC	
mean	0.1353
std	0.0844

(5-11)

From the histograms shapes, we can observe that the three Agents have a control and a noise rejection behaviour comparable to the Expert. The Exact policy is very close to the Expert, while Subgradient and MP ones are slightly different. Interestingly, even if mean and std. od Subgrad. Agent are larger than the Expert ones, the histogram shape qualitatively appears to be better. In fact, there are more instances of low cost values (range 0-0.08) for the Subgradient Learner, meaning that it managed, more often than the Expert, to keep the stage cost very low. However, there is a longer tail and some instances of quite larger cost values (outside the plot limits) that lead to overall larger mean and std. Very interestingly,



(a) Exact Agent

(b) Subgradient Agent

(c) MP Agent

Figure 5-3: QR Cost histograms, with mean and std. (Learner vs Expert)

the MP Agent always achieves a better histogram than the Expert MPC, with a similar shape to the Subgradient one (with more instances of low-cost values), but without the longer tail. One of the reasons of this surprising MP performance is that it has generally lower tracking errors for y_2 and y_4 (as previously observed), resulting in lower QR costs over the experiments.

Let us draw some conclusions. All the Learning Agents are successful in their task, with a performance comparable to the Expert. The Exact Agent manages to precisely fit the available Expert data in the learning process (pushing very accurately the numerical values of the parameter), resulting in a policy that is a precise approximation of the Expert one, which ultimately is the goal of Inverse Learning. On the other hand, the two iterative approaches, achieve a more different policy, wrt. to the Expert, and a sub-optimal solution wrt. the Exact Agent (with a larger final sub-optimality loss value). However, the learning process follows a more fluid and natural evolution, improving θ estimates at each learning iteration. Ultimately, the result is a numerically more even parameter matrix, which contains a larger approximation of the exact Expert policy. In this difference lies perhaps the key to a more general policy, which captures more naturally (wrt. its own mathematical formulation) the nature of the problem and the info underlying the training set. If the Exact Agent pushes the numerics and force them to exactly adapt its policy to the Expert one (as an analogy, we could imagine it as a sort of over-fitting behaviour), the iterative algorithms evolve in a

certain sense more freely. In fact, we can observe how their final policy achieves histograms which are slightly different in shape and even better than the Expert one.

5-2 Changing reference, richer feature, anticipative action

At this point, we complicated the control task and the Learner structure, in order to test its capabilities in a changing reference scenario, which was explained in Section 4-3. In this scenario, additionally to the In-Sample test, we conduct five different Out-of-Sample tests, reshuffling (in the last four of those) the reference signal, and using a different dynamics noise signal. This way, we can be sure that the our Agent actually learned a general and solid policy, as we test it in many possible configurations. The Subgradient algorithm uses $\eta = 0.005$ and 5000 learning iterations, while MP $\eta_a = 0.0001$ and $\eta_b = 0.1$. We also conducted the same experiments with absence of noise, achieving very similar results and conclusions.

Regarding the In-Sample test, both Exact and Subgradient Agents disagree with the Expert in a similar way, with comparable accuracy values. These are lower than in the simpler Single Reference scenario, as now the learning and control task is more complex. It is thus expected that the learned policy is a less precise approximation of the unknown true Expert policy and acts differently (see Table 5-12). Additionally, in this scenario, the MP Agent has a lower accuracy and, as we will soon show, a worse Out-of-Sample performance. That is due to the fact that the MP learning process depends on the learning rates η_a and η_b and, in this scenario, the best tuning we managed to achieve leads to a slightly worse (but still comparable) performance.

	Exact	Subgrad.	MP
agrees	1079	1045	978
accuracy	57.55%	57.73%	52.16%

(5-12)

From the Out-of-Sample test we can again observe how the performance of the Learner Agents is comparable to the Expert, which is quite remarkable given the complexity of this specific learning/control scenario. It is interesting to observe how the Exact Agent achieves a lower Tracking Error for y_2 w.r.t the MPC Expert, in all the experiments.

Tracking Errors in the multiple tests are:

$$TrackErr_{Exact} = \begin{array}{|c|c|c|c|c|} \hline 1 & 2 & 3 & 4 & 5 \\ \hline 100.3788 & 104.3093 & 100.1555 & 101.3809 & 100.3788 \\ 111.7046 & 112.4425 & 112.0170 & 112.1990 & 111.7046 \\ 375.1426 & 375.1604 & 362.5594 & 369.1937 & 375.1426 \\ 200.6154 & 203.8956 & 193.9634 & 204.2039 & 200.6154 \\ \hline \end{array} \tag{5-13}$$

$$TrackErr_{Sub} = \begin{array}{|c|c|c|c|c|} \hline 1 & 2 & 3 & 4 & 5 \\ \hline 110.1931 & 108.6264 & 105.3377 & 108.5360 & 106.7798 \\ 123.8993 & 125.0751 & 122.5186 & 121.8642 & 120.9716 \\ 358.4859 & 362.1561 & 350.7620 & 369.4598 & 351.6030 \\ 193.6561 & 197.2754 & 191.9796 & 197.7127 & 191.5908 \\ \hline \end{array} \tag{5-14}$$

$$TrackErr_{MP} = \begin{array}{|c|c|c|c|c|} \hline 1 & 2 & 3 & 4 & 5 \\ \hline 113.2788 & 115.6181 & 110.5932 & 113.5231 & 111.4633 \\ 127.6486 & 127.1125 & 124.5042 & 126.2333 & 123.0893 \\ 336.3165 & 357.0837 & 353.2837 & 356.4193 & 352.4305 \\ 200.4757 & 201.5596 & 197.7019 & 200.1776 & 198.8591 \\ \hline \end{array} \tag{5-15}$$

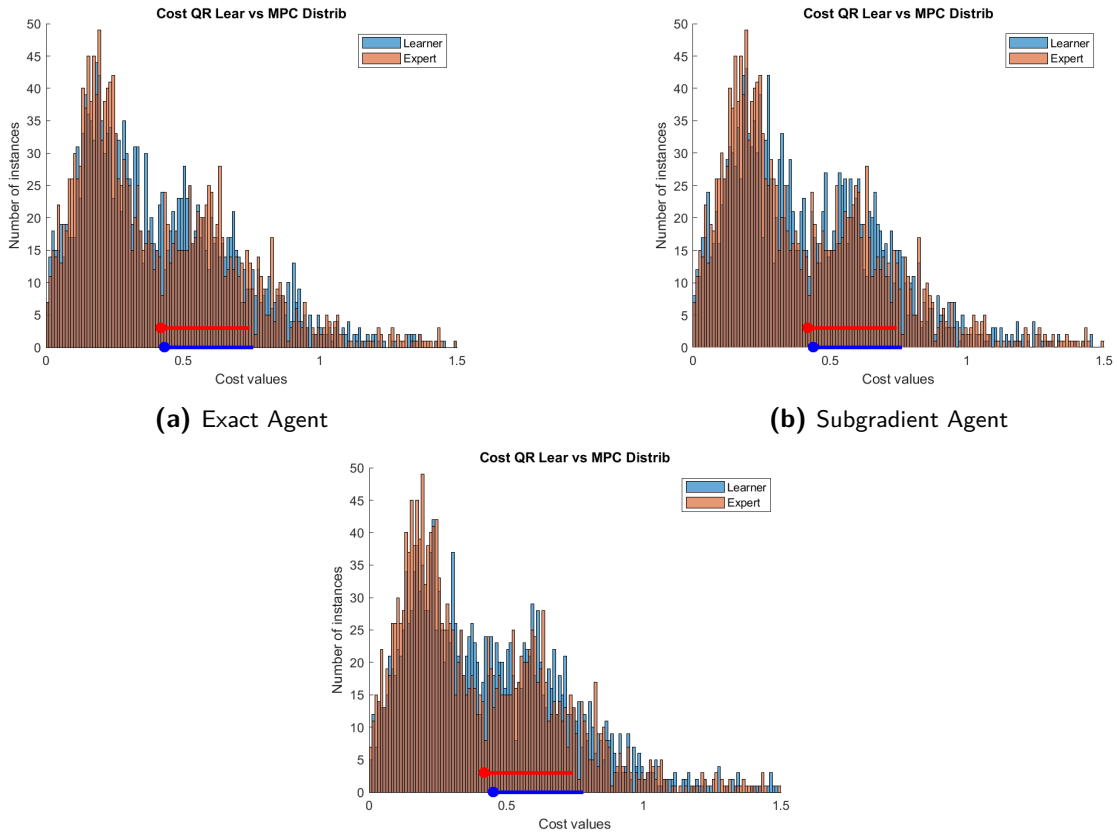
$$TrackErr_{Expert} = \begin{array}{|c|} \hline 99.5705 \\ 115.5188 \\ 359.9951 \\ 191.2546 \\ \hline \end{array} \tag{5-16}$$

In Fig. 5-4 it is possible to observe the QR cost histogram for each Agent (compared to the Expert one), achieved in the second Out-of-Sample experiment. Again, it is the most challenging of the five experiments, due to the specific numerics (here given by the specific reference signal), and it yields the worst result from the five. It is clearly noticeable how the histograms of our Learner Agents approximate well the Expert behaviour and noise rejection properties, with a similar shape and mean and standard deviation (std).

Interestingly, in the other four experiments (1,3,4 and 5), the Exact Agent achieves a better histogram, with lower mean and standard deviation (see Tables 5-17 to 5-20). Observing the Tracking Errors in Tables 5-13 and 5-16 it is possible to notice how the Exact Agent performs a more precise tracking of the second output, while being slightly less precise on the first one. But in doing so, with this slightly different control approach, achieves an overall better histogram. In two experiments also the Subgradient Agent achieves that result (and both Subgradient and MP Agents various times achieve a lower std.).

That, again, highlights the high potential of the Inverse Learning Agents and the fact that they can achieve a control policy that captures well the Expert control behaviour, while being different (as concluded from the In-Sample testing). Actually, that difference can lead sometimes to a better performance, as the achieved policy can capture (through the feature space and the parametrization achieved for that space) more deeply and in a more concise way system evolution and control behaviour, underlying in the available data-set.

Furthermore, the use of a parameter that exploits a feature space allows to synthetically capture the noise rejection properties. In fact, instead of needing to solve at each time step



(a) Exact Agent

(b) Subgradient Agent

(c) MP Agent

Figure 5-4: QR Cost histograms (Learner vs Expert)

a complex optimization program that cannot exactly take into account the random dynamics noise (as the Expert MPC does), our Agents directly absorb, in the learning process, the information connected to the noise present in the training data-set. Observing an already perturbed system evolution trajectory and the many peculiar system configurations encountered with that specific type of noise, it tries to fit the parametrized policy to that information, managing to naturally achieve its own control policy, able to effectively deal with the noise.

QR Exact	1	2	3	4	5
mean	0.4125	0.4318	0.4031	0.4142	0.4125
std	0.3064	0.3190	0.3042	0.2819	0.3064

(5-17)

QR Sub	1	2	3	4	5
mean	0.4209	0.4381	0.4140	0.4221	0.4137
std	0.2950	0.3193	0.3058	0.2963	0.2998

(5-18)

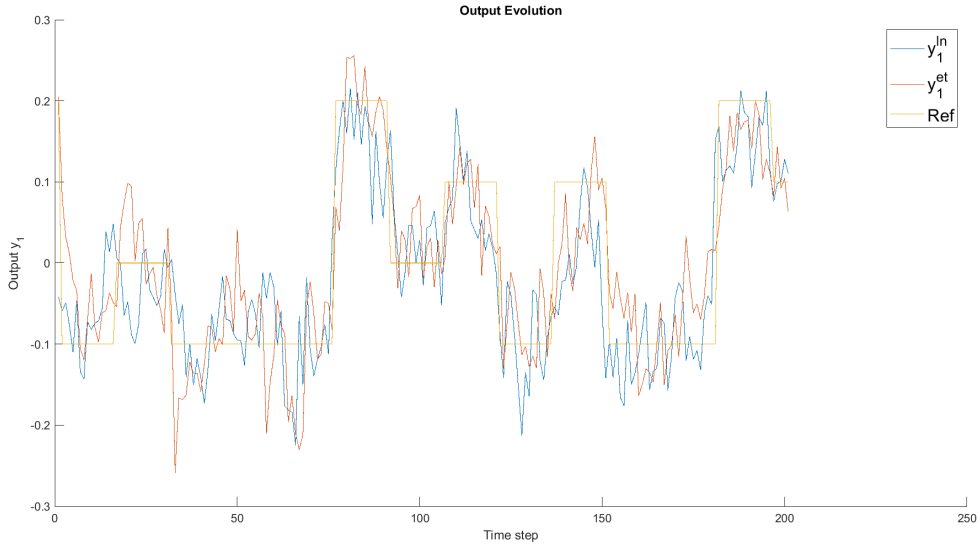
QR MP	1	2	3	4	5
mean	0.4389	0.4535	0.4302	0.4357	0.4318
std	0.2990	0.3205	0.3095	0.2988	0.3095

(5-19)

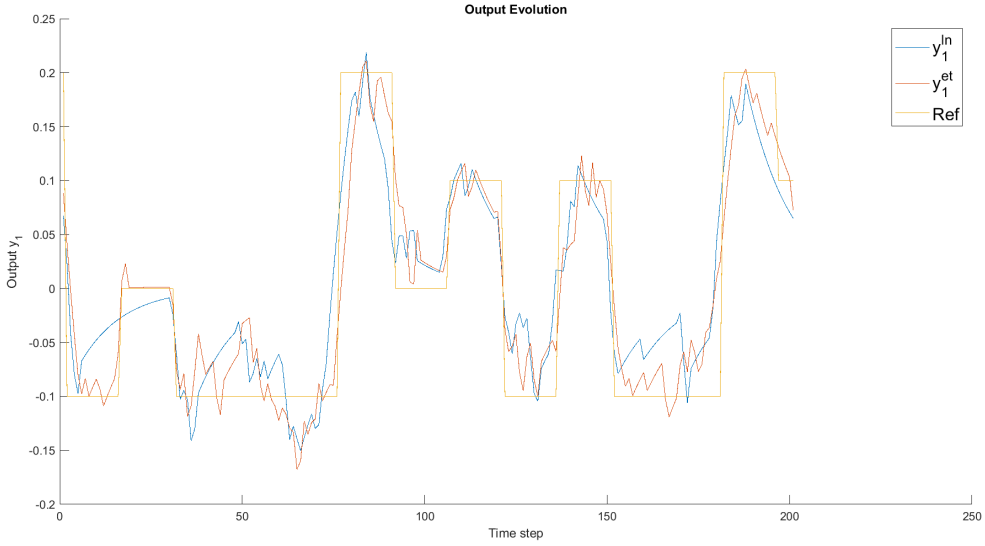
QR MPC	
mean	0.4182
std	0.3187

(5-20)

In general, from the numerous numerical experiments, we can conclude that the Learner performance is comparable to the Expert and meets the desired requirements. Additionally, with a better tuning of the MP Agent it would be possible to improve its performance, as in this scenario it is slightly undertone wrt. the other two Learner Agents. Anyhow, in all the achieved results (for both Single and Changing Reference scenarios), the differences between the several Agents are minimal. We carried out an analysis on the differences and on the reasons thereof based on this small differences. But, in broader terms and on a more high-level analysis, the Agents' performance is almost identical. In Fig. 5-5a it is possible to observe an excerpt of the first output tracking evolution, in the presence of dynamics noise, while in Fig. 5-5b the same in the scenario without noise.



(a) Exact Agent vs Expert (with dynamics noise)



(b) Exact Agent vs MPC (without dynamics noise)

Figure 5-5: Exact Agent vs MPC (y_1)

5-3 Learning process: analysis and comparisons

After having assessed the performance capabilities of various Learner Agents (for the Offline Learning case), we will now analyze more in depth the learning process itself. Mainly, we are interested in comparing the convergence speed and accuracy of the three different learners.

First of all we analyze the **Exact Learner**. We train this Agent with an increasing number of training data-samples, monitoring the time needed to complete each program. Obviously, the needed time increases with the size of the training data-set. We can additionally observe something interesting: the required optimization time increases linearly up to around 11000 data samples, when it surges (about 100 seconds) and subsequently keeps increasing linearly with a higher slope. The exact values of slopes and surge position probably depend on the hardware used for the experiments, but we can in general expect such a behaviour. In fact, up to a certain number of data samples, the solver manages to allocate all the data in faster memory locations and access, evaluate and process it quickly, as well as maybe apply some pre-processing and data-handling strategies that can speed up the whole process and make the required time predictable. When the data-set becomes too large, data handling by the solver becomes slower (as probably various slower memory regions must be exploited) and we observe that surge in the computation time. When that happens, as the computation becomes more involved, required time keeps increasing with a higher slope. That can be observed in Fig. 5-6, in which the Exact Learner is being trained on Oil Fractionator (Single Reference scenario) Expert data.

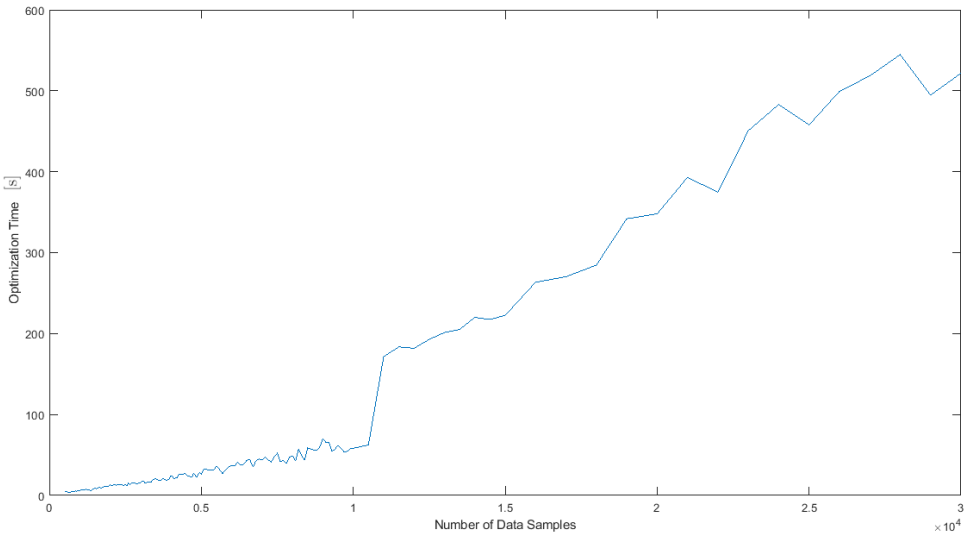


Figure 5-6: Data samples vs optimization time for the Exact Learner

Let us now compare the various learning performances of the iterative algorithms. In particular, we will compare MP with the Subgradient Descent algorithm. We compare the sub-optimality loss values of these Learners over the number of iterations and over time as well . To each of those we subtract the Exact optimal loss value (achieved by the Exact

Agent), so that we have a direct numerical indication about how close each Agent gets to the best achievable result.

Let us begin with the Oil Fractionator Single Reference scenario and a training-set made of 1000 data points. Subgradient "Full" and MP exploit all the available data points at every iteration. On the other hand, each Subgradient "Batch" algorithm uses a randomly selected batch of b data-points among the available ones. We performed a manual fine-tuning of the learning rate η for each Agent, in order to try to observe the highest potential of each of them. However, there might exist a better tuning for each Agent. Also, the tuning depends on the System/Control scenario, amplitude of the data set and used algorithm, so it is complex to guarantee the best possible tuning in each scenario. Nevertheless, we get some interesting indications from the achieved plots (Fig. 5-7).

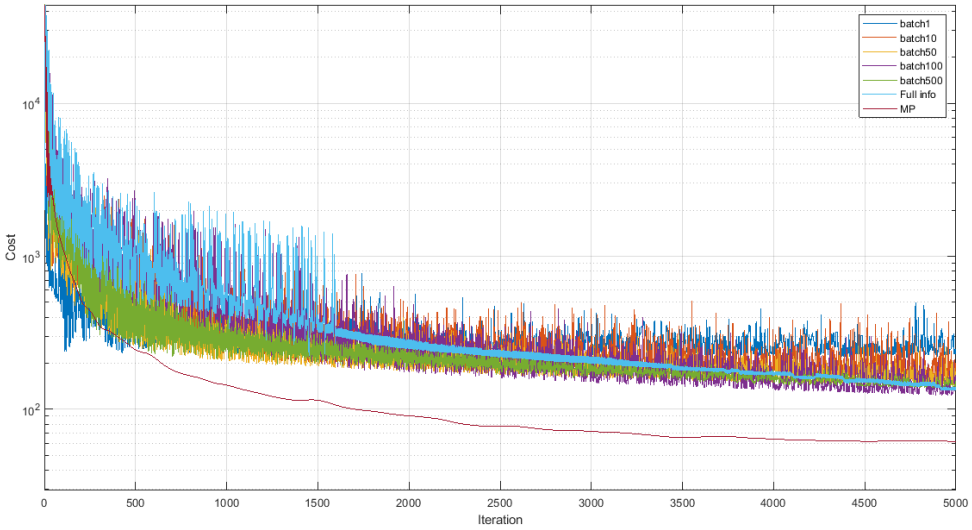
Regarding the "batch" Agents, we compute the cost at every step over the whole data-set (1000 points) and not only over the batch, in order to have comparable and meaningful values. Regarding the Time-Loss plot, the cost is being computed every two seconds, for 400 seconds. As the various Agents have different computation times, we have different number of performed iterations:

$$\text{Iterations} = \begin{array}{|c|c|c|c|c|c|c|} \hline \text{batch 1} & \text{batch 10} & \text{batch 50} & \text{batch 100} & \text{batch 500} & \text{Full} & \text{MP} \\ \hline 1160159 & 565509 & 201483 & 114037 & 31379 & 16268 & \sim 5000 \\ \hline \end{array} \tag{5-21}$$

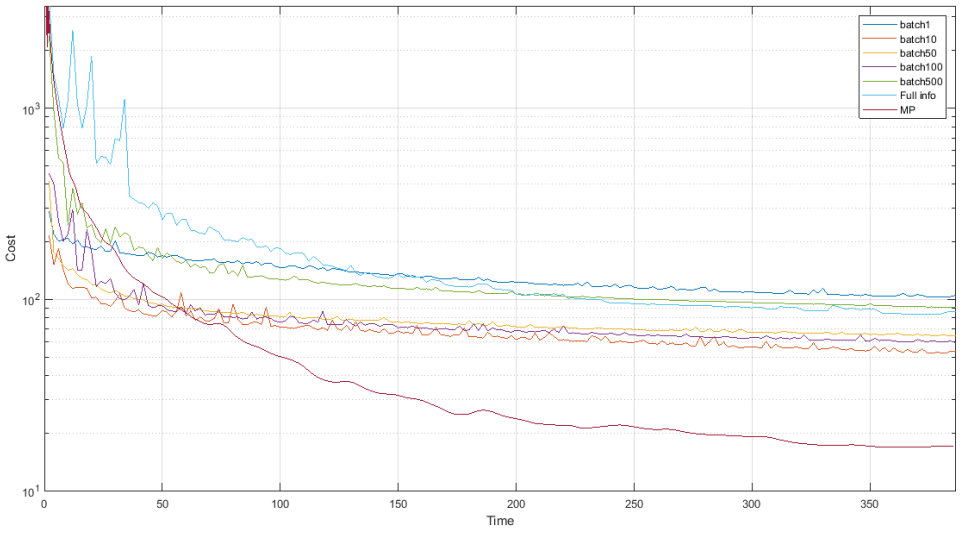
The learning rates we use are:

$$\eta = \begin{array}{|c|c|c|c|c|c|c|} \hline \text{batch 1} & \text{batch 10} & \text{batch 50} & \text{batch 100} & \text{batch 500} & \text{Full} & \text{MP} \\ \hline 0.08 & 0.05 & 0.01 & 0.008 & 0.001 & 0.001 & \begin{array}{c}\eta_a = 0.002 \\ \eta_b = 0.01\end{array} \\ \hline \end{array} \tag{5-22}$$

We can observe how the MP learner manages to get much closer to the optimal admissible loss values and in less iterations. Even in the Time-Loss plot (Fig. 5-7b), it outperforms the Subgradient Agents (even though it requires more time to complete each learning iteration). From the plots we can also notice that the Subgradient batch variants are performing quite well. They are generally more noisy in the learning process but faster in the computation at every iteration. In fact, in the Time-Loss plot, we observe that they actually learn quicker than the "Full" Subgradient algorithm, and even get closer to the the true optimum. Again, that may slightly depend on the specific tuning and problem setting and the time we allow for the learning process, but in general the plot showcases the potential that a batch Agent has, especially if we are dealing with a large data-set.



(a) Loss over algorithm iterations



(b) Loss over time in seconds

Figure 5-7: $l^{sub} - l^{Exact}$ for various Learners

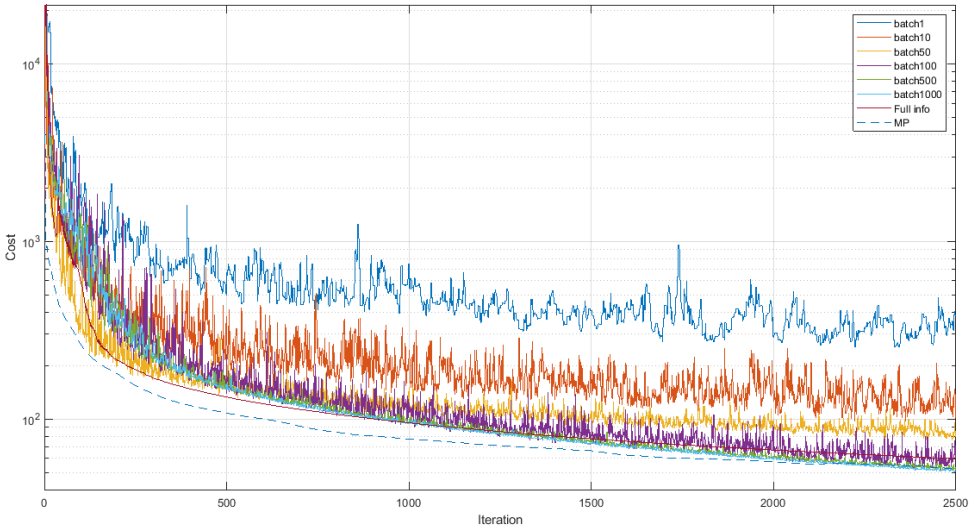
Let us now perform the same experiments in the Oil Fractionator Multi-Reference Tracking scenario. We perform the learning process with both Sugradient Descent (Full/Batch variants) and MP algorithms on an expert training set of 1875 data-points (see Fig. 5-8). The learning rate tuning we use are:

$$\eta = \begin{array}{|c|c|c|c|c|c|c|c|} \hline \text{batch 1} & \text{batch 10} & \text{batch 50} & \text{batch 100} & \text{batch 500} & \text{batch 1000} & \text{Full} & \text{MP} \\ \hline 0.5 & 0.2 & 0.05 & 0.05 & 0.01 & 0.005 & 0.002 & \begin{array}{c}\eta_a = 0.0005 \\ \eta_b = 0.1\end{array} \\ \hline \end{array} \tag{5-23}$$

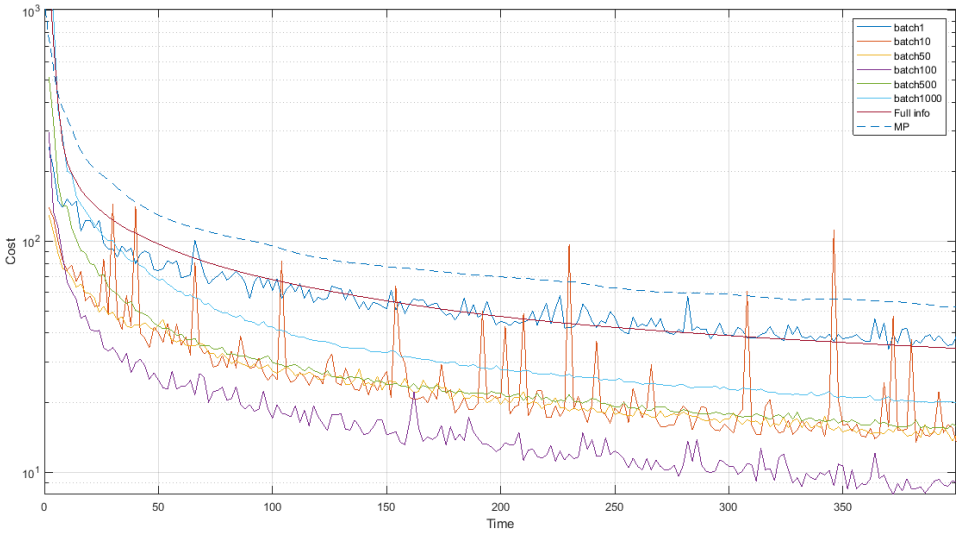
From the Iterations-Cost plot (Fig. 5-7a), we can observe how MP and Full Subgradient Descent have a very close behaviour, followed by the batch Subgradient variants. However, they are slower than the batch variants in the computation and that does not go unnoticed here, with a larger data-set (1875 data-points, each with a larger feature vector). In fact, in the Time-Cost plot (Fig. 5-8b), we see how the batch algorithms get lower sub-optimality loss values much quicker, as they perform many more iterations than MP/Subgradient 'Full' (under the same time interval). Additionally, in Fig. 5-7a, it is clear how learning processes of low batches ($b = 1$ and $b = 10$), in addition to having a much more noisy evolution, converge to higher loss values, meaning that the learned policy is a less precise approximation of the objective Expert (that is especially the case in this Changing Reference Scenario). That happens because each learning descent step depends on the Subgradient computed on that small amount of data, randomly sampled. As we are dealing with a complex data-set containing a variety of situations (different changing references), randomly sampling a few data-samples (which is somehow taking them out of context) at each iteration and using them to determine the learning direction (Subgradient) will lead to imprecise and ever changing learning directions, failing to precisely converge to the desired parameter θ.

From these observations, we can draw some conclusions for the offline learning scenario, also taking into consideration the results from In/Out-of-Sample testing. If we are dealing with a training data-set with a reasonable size, we suggest using the Exact program or the MP Agent. The latter however has a more difficult tuning process, as there are two learning rates and their tuning is quite relevant. The Exact Agent will provide the closest possible policy to the Expert, while the MP could provide a slightly different one (which however captures well the main aspects of the control task), which can sometimes yield benefits.

On the other hand, if we are dealing with a more complex problem and/or larger data-set (as it is the case for the Changing Reference Scenario in our case study), then Batch Subgradient Descent is recommended, with its much faster computation times for each iteration. And with a sound batch (should have a proper size, in order for it to being able to capture meaningful information at each learning iteration) and learning rate choice, the result is comparable to the other approaches.



(a) Loss over algorithm iterations



(b) Loss over time in seconds

Figure 5-8: $l^{sub} - l^{Exact}$ for various Learners, Multi-Ref scenario

Chapter 6

Numerical results - Online Learning

In the Online scenario, the Learner is directly acting on the system while performing the learning process and improving its control policy, based on Expert corrective advice (see Fig. 6-1). In fact, at each time step k, the Expert Agent computes its optimal input and provides that information to our Agent, which will be able to exploit it with one step delay, to mimic actual applications, where there are information transmission delays. The Expert is kept active until a moment in time T_{cut}, after which it is shut off, the iterative learning process ends and the Learner keeps controlling the system with what he learned up to that moment, with the latest θ estimate.

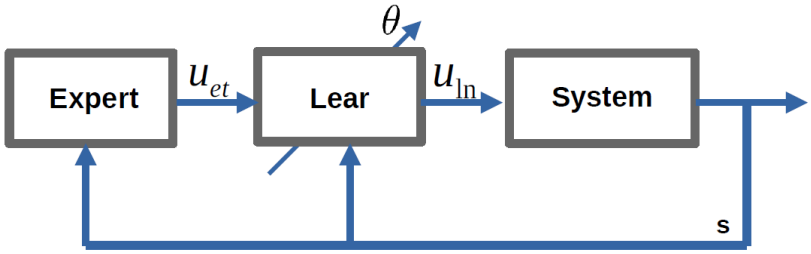


Figure 6-1: Online Learning Scheme

Subgradient Learner Agent In order to learn the paremeter θ online, we use the batch Subgradient Descent algorithm. At each time step, the Learner Agent computes the Subgradient w.r.t. θ exploiting the last b (with b batch size) measured data-points. However, as the parameter is being updated at each time step, the Learner policy is also evolving and improving with time. It means that our Agent, according to its most recent policy, would have acted differently in the previous moments in time, when that policy was different. In fact, as his control action u_θ^{ln} is achieved minimizing the cost $F_\theta(s,u)$, it is clear that, in general, $u_{\theta,k}^{ln} \neq u_{\theta,k-i}^{ln}$ (with θ,k indicating the θ estimate at time-step k, while $\theta,k-1$ the estimate at time $k-i$). Let us recall that the batch Subgradient computation requires the availability of $\{s_i, u^{et}(s_i), u_\theta^{ln}(s_i)\}$, with $i = k-b,...,k$. That is clear from the following equation, which

shows the Subgradient computation at the current time-step k of the online learning/control process:

$$\nabla l^{sub} = \sum_{i=k-b}^{k} \begin{bmatrix} s_i \\ u^{et}(s_i) \end{bmatrix} \begin{bmatrix} s_i \\ u^{et}(s_i) \end{bmatrix}' - \begin{bmatrix} s_i \\ u^{ln}_{\theta,k}(s_i) \end{bmatrix} \begin{bmatrix} s_i \\ u^{ln}_{\theta,k}(s_i) \end{bmatrix}' \tag{6-1}$$

For that reason, because of the ever changing policy, at each time step the Agent must re-compute the optimal inputs $u^{ln}_{\theta,k}(s_i)$ over the data-points s_i contained in the batch (with $i = k-b, ..., k$), using its current parameter estimate.

As the Learner Agents is acting on the system from the beginning, when he is just starting the learning process and does not have a solid cost parametrization, he will initially tend to apply sub-optimal control inputs and the system evolution trajectory might deviate (even significantly) from the desired behaviour and break the constraints.

For that reason, we introduce a "safety feature", that limits the initial deviation from the desired behaviour, by computing the optimal control input solving a constrained optimization program. In fact, we minimize the cost function $F_\theta(s, u)$ only over the input values that guarantee the respect of the imposed constraints. That is possible exploiting the knowledge of the system LTI model, as for each possible input vector we can predict the system state and output vectors at the subsequent time-step ($x_{k+1} = Ax + Bu$ and $y_{k+1} = Cx_{k+1} + Du_{k+1}$). Recalling that we are solving the program with a combinatorial approach, we take into account only the input vectors that respect the constraints $y_{min} \leq y_{k+1}(u_k) \leq y_{max}$. The constraint is then simply incorporated in the set $\mathbf{U}(s)$, for each possible feature vector s. For that reason, at time-step k of the online learning/control scenario, the Learner Agent is acting according to the policy computed through the program:

$$u^{ln}_k = \arg\min_{u \in \mathbf{U}(s_k)} F_{\theta,k}(s_k, u) \tag{6-2}$$

Exact Learner Agent An additional learning approach that we follow is to exploit the **Exact** θ solution: at every time step the Learner Agent solves one unique optimization program, taking into account all the previous data (see Section 3-2). So, instead of iteratively (and sub-optimally) update the θ estimate as the Subgradient method does, it achieves at every time-step the true optimal parametrization, w.r.t. to the available information. And, as expected, the performance is much more accurate, even better than the MPC Expert Agent, which is quite surprising (results are in Sections 6-1 and 6-2).

That approach shows the highest potential of the Inverse Learner Agent, as it exploits, at best of its possibility, the available information. Such a formulation, however, may not always be applicable in reality, as at each step a very large and long optimization program must be run, especially for the later moments of the simulation (or actual application on the plant), when thousands of data-points are taken into account for example (as it would take too long to complete). In our specific case, the largest program is solved at $T_{\text{cut}} = 2000$ and it takes into account all the available data points $\{\hat{s}_k, \hat{u}^{et}_k\}$ with $k = 1, ..., T_{\text{cut}}$. At time-step T_{cut} it takes 50 seconds circa to complete (with the combinatorial approach which ensures fast

computation times), so it would be even applicable in reality, as the sampling time is four minutes. Furthermore, we could be able to sustain such an approach up to a larger T_{cut} moment in time and consider even more data-points for the learning process.

Nevertheless, the Exact program could actually be exploited in very large problems (larger MIMO systems, with more control inputs and a much larger integer set of admissible values for these inputs) and shorter sampling times. In fact, if we run this long program every now and then, only when we desire to improve our controller, and in the meantime use the last available parametrization for controlling the system, then the long computation times are not an issue anymore. In fact, in the first dozens of time-steps, the program has reasonable computation times and can be run at each time step (as in our simulation). This way, we achieve rapidly a valid θ parametrization, that can effectively control the system (or we could in the beginning use an initial θ estimate achieved through an offline program for instance). Then, every T_{update} steps, we can run in background a very large program, that can take many time-steps to compute, while the previous available parameter is being used in order to control the system. When the program is completed, we update our parameter with the new improved solution. In conclusion, even the Exact solution algorithm can be used in the online control scenario and here we show the high potential that such an approach can yield.

Online Learning approach strengths Online Learning takes advantage of a stronger and more informative feedback signal. In fact, as the Learner is directly acting on the system and guiding the output trajectory evolution, he receives tailored Expert corrective advice for its own behaviour. Now, he is not simply observing an Expert data-set and trying to fit it, but he is learning by doing, exploring more the feature-space and leading the system in many more configurations (depending both on the specific evolution of the learning parameter θ and on the current system configuration). Each time, he receives advice from the Expert, for the specific current situation, which makes the learning process more natural and meaningful.

That aspect is analyzed and addressed in [LKYC16], for Imitation Learning tasks. In general, in the supervised learning setting (and more in general, in offline learning tasks), the assumption is that all the data (training and validation) is being sampled i.i.d. from the same data distribution D. This way, we are sure that the learned hypothesis class (or, in our case, control policy) is able to generalize properly. In our scenario, it would mean that all the data-points (so feature and input vectors) used for learning and control are sampled from one distribution ($(s, u) \sim D$). However, as we are not simply learning a hypothesis class from a generic data-set, but a control policy for a dynamical system, which later needs to be controlled, the problem does not respect that assumption anymore. In fact, in the Offline Learning scenario, we have an expert data-set, that contains data-points from Expert trajectories, achieved when applying the Expert policy π^{et}. This can mathematically be expressed as $s \sim D_{\pi^{et}}$, where the distribution $D_{\pi^{et}}$ indicates all the available Expert data-points, from which we can achieve a subset for our training purposes and achieve a certain approximated learned policy π^{ln}. However, when we apply our policy on the system (Out-of-Sample), it will lead the system on new trajectories and new samples of feature signal s, which are not contained in the original distribution $D_{\pi^{et}}$. Actually, in the testing phase, we are sampling from

a different distribution, having $s \sim D_{\pi^{ln}}$ and breaking the initial supervised learning assumption of i.i.d. sampling. For that reason, we cannot be sure that the learned policy π^{ln} will generalize properly and achieve the desired performance. It may even happen that the learned control policy leads the system to a failure. In fact, as it differs from the Expert policy, it will apply different inputs and lead the system on a different trajectory. If that trajectory deviates too much from the one observed in the training set (and contained in the distribution $D_{\pi^{et}}$), our Agent must act in completely new situations and, in complex tasks, it may fail completely.

A possible solution (proposed in [LKYC16]) is to complete multiple rounds of offline learning, generating a new trajectory for each intermediate Learner policy and new Expert data on that trajectory. Otherwise we could exploit multiple Expert data-sets on various trajectories. An alternative (which we follow here), is the Online Learning approach. In that scenario, all the observed data-points are sampled from the same set (or distribution) of all the points reached by the learned policy, as the Learner is directly controlling the system while learning. Learning is possible with real-time Expert corrective advice, which contains reinforced and situation-tailored information.

6-1 Online single reference control scenario

In this control scenario (see Section 4-2 for details), the Learner Agent is controlling the system (with gaussian dynamics noise acting on it) while receiving Expert corrective advice until time-step $T_{\text{cut}} = 2000$. At that point, the Expert is cut out of the loop and our Agent keeps controlling the system for other 2000 time-steps, exploiting the policy given by θ_{Tcut}, with a new gaussian noise sequence. The parameter θ is initialized at zero, so the Learner Agents starts without any knowledge nor control policy. We experimented both with the batch Subgradient Agent (with batch sizes $\{1, 10, 50, 100, 500\}$, each with its own fine-tuned learning rate) and with the Exact Agent.

Obviously, as during the initial learning phase (until time-step T_{cut}) the control is less precise, we observe larger Tracking Errors and worse QR histograms. After that point, the Expert is cut out from the loop and the learning process is ended. Now, the Learner controls the system on its own, achieving a performance comparable to the Expert. We provide all the numerical results both for the time span $(1 : T_{\text{cut}})$ (let us refer to it as "phase 1") and for $(T_{\text{cut}} : T)$ (we refer to it as "phase 2"), in order to additionally showcase how, even in the initial learning phase, the performance is satisfactory and, in certain cases significantly close to the Expert.

We provide Tracking Errors (one for each output) and QR cost histograms mean and std., both for the Expert (MPC) Agent and for the Exact Learner Agent. For the latter, as there are two experimental phases,$(1 : T_{\text{cut}})$and $(T_{\text{cut}} : T)$, we provide two tables:

$$Expert_{(1:T_{cut})} = \begin{array}{|c|c|c|}\hline \text{TrackErr} & \text{QR}_{mean} & \text{QR}_{std} \\ \hline 74.5717 \\ 98.0419 \\ 316.9342 \\ 153.3126 \\ \hline \end{array} \quad \begin{array}{|c|c|}\hline \text{QR}_{mean} & \text{QR}_{std} \\ \hline 0.1476 & 0.0930 \\ \hline \end{array}$$

$$Exact_{(1:T_{cut})} = \begin{array}{|c|c|c|}\hline \text{TrackErr} & \text{QR}_{mean} & \text{QR}_{std} \\ \hline 76.9778 \\ 79.9187 \\ 286.3439 \\ 162.6731 \\ \hline \end{array} \quad \begin{array}{|c|c|}\hline \text{QR}_{mean} & \text{QR}_{std} \\ \hline 0.1268 & 0.1173 \\ \hline \end{array}$$

$$Exact_{(T_{cut}:T)} = \begin{array}{|c|c|c|}\hline \text{TrackErr} & \text{QR}_{mean} & \text{QR}_{std} \\ \hline 74.9102 \\ 75.5072 \\ 280.7285 \\ 157.3974 \\ \hline \end{array} \quad \begin{array}{|c|c|}\hline \text{QR}_{mean} & \text{QR}_{std} \\ \hline 0.1147 & 0.0725 \\ \hline \end{array}$$

It can immediately be observed that the Exact Agent performs better than the Expert, already in the initial learning phase, achieving a better control policy. In fact, it exhibits a much lower Tracking Error for y_2 (79.9187 in the time span $(1 : T_{\text{cut}})$ (phase 1) and 75.5072 in $(T_{\text{cut}} : T)$ (phase 2), against Expert's 98.0419). It has a lower QR mean in both cases, and a significantly better histogram in $(T_{\text{cut}} : T)$, with mean valued 0.1147 and std. 0.0725, against Expert's mean 0.1476 and std. 0.0930. It is a very good result and it shows the full potential of the Inverse Learning Agent. However, the comparison is not completely fair in this case, as the MPC Agent has at each time step a limited horizon for computing its control inputs, while the Exact Agent used here has full past information (so more information). Nevertheless, it is surprising that it manages so clearly to outperform the Expert, as it is learning from its corrective advice to behave as closely as possible as him.

Regarding the Subgradient Agents, there are multiple results, as we conducted experiments with different batch sizes. We provide a dedicated table to the Tracking Errors and one for QR cost histogram values (each column corresponds to a specific batch size). As there are two experimental phases, there are two instances of these tables:

$TrackErr^{Sub}_{(1:T_{cut})} =$

b	1	10	50	100	500
η	0.007	0.007	0.01	0.05	0.05
	111.8827	122.4270	115.4332	88.2842	88.6824
	145.2402	176.0181	167.0744	115.7374	99.8887
	372.0379	378.5168	322.2350	293.6263	291.0106
	216.2525	225.3624	193.6257	169.6027	166.8112

$QR^{Sub}_{(1:T_{cut})} =$

b	1	10	50	100	500
η	0.007	0.007	0.01	0.05	0.05
mean	0.2952	0.3517	0.3217	0.1726	0.1527
std.	0.4790	0.4934	0.5619	0.1743	0.1134

$TrackErr^{Sub}_{(T_{cut}:T)} =$

b	1	10	50	100	500
η	0.007	0.007	0.01	0.05	0.05
	100.9531	97.2700	90.3099	78.5429	80.3522
	163.8271	133.7249	128.1831	97.0258	88.5853
	245.9343	251.5622	247.2706	287.1671	290.9766
	170.8115	148.7261	159.1901	161.6885	167.5203

$QR^{Sub}_{(T_{cut}:T)} =$

b	1	10	50	100	500
η	0.007	0.007	0.01	0.05	0.05
mean	0.2375	0.1883	0.1748	0.1336	0.1322
std.	0.1961	0.1395	0.1270	0.0869	0.0885

Regarding the batch Subgradient Agent, from both Tracking Errors and QR histogram values, in the two experiment phases, we can clearly observe that performance increases with the size of the used batch. Batches 1 and 10 achieve a worse performance than the Expert, in both the experiment phases (but especially in phase 1, when learning is progressing). As we discussed in Section 5-3, such small batches cannot follow a meaningful and consistent learning direction (with the Subgradient), resulting in a noisier and less precise learning process. Here additionally dynamics noise perturbing the system and limited learning time (up to time-step T_{cut}) are to be taken into account. Anyhow, from observing the output plots, we observed how the desired control policy is being approximated and the general behaviour follows the desired one. While the control is less precise (especially for the y_2), the quite larger Tracking Errors also depend on the fact that the weighting matrix $Q = diag(20, 10, 0, 1)$ contains large values, so small consistent deviations from the reference value are heavily penalized.

On the other hand, batches 100 and 500 achieve a comparable performance to the expert already in phase 1, w.r.t. both to Tracking Errors and QR histogram values. Performance is additionally improved in phase 2, when the learning process is completed. In fact, in phase 2, these two Agents yield a better QR histogram than the Expert one, due to the fact that they control more precisely the second output. The same analysis done in the offline case holds: Subgradient Agent (with the proper batch size) learns its own policy, slightly different from the Expert but capturing the sense of the control behaviour, sometimes yielding a better performance. Here, that result is even stronger, as we benefit from a stronger feedback from

which to learn, as the corrective advice is being provided on the Learner trajectory, while it is exploring the feature space on its own. The additional aspect of a limited complexity control task (Single Ref. Tracking) and that the learning phase lasts a sufficient amount of time (2000 time-steps) allows for a very efficient learning.

6-2 Online changing reference control scenario

In this scenario, the Learner Agent learns/controls the system over the original reference sequence, lasting 1875 steps (125 possible reference vectors, each lasting 15 steps). When the sequence is terminated, we cut out the Expert MPC from the loop, meaning that $T_{\text{cut}} = 1875$. At this point, our Agent controls the system on the same reference sequence (for other 1875 steps) and after that on a reshuffled reference sequence, which we now refer to as phase 3 (for other 1875 steps). This way we achieve solid performance indicators, as we can directly compare phase 1 with phase 2 on the same reference signal, as well as test how our Agent generalizes on a different reference. All the results for the reshuffled reference are indicated with the sub-script "Resh.", we refer to them as phase 3 of the experiment, and will have an additional dedicated table.
This scenario is much more complex. In fact, not only there is a much larger feature space and a complex control task, but during the online learning process (phase 1), our Agent observes each reference value (and the corresponding Expert corrective advice) for only 15 time steps. And the learning time is limited to $T_{\text{cut}} = 1875$. In this experiment, we removed the dynamics noise from the loop, in order to observe cleaner results, as the task is already quite challenging. However, as we have seen before, noise does not jeopardize the learning process. So even in this scenario, it would simply complicate a bit the learning process, deteriorate to a certain degree the performance and, in the Subgradient Agent case, make its learning evolution noisier.

Let us analyze the experimental results. As in the Single Reference Online scenario, we provide Tracking Errors (for the four outputs) and QR values in a Table, both for the Expert MPC and the Exact Agent (phase 1,2 and 3):

$\text{Expert}_{(1:T_{cut})} =$

TrackErr	QR_{mean}	QR_{std}
69.7523	0.3575	0.2811
98.3625		
350.2278		
191.4000		

$\text{Exact}_{(1:T_{cut})} =$

TrackErr	QR_{mean}	QR_{std}
76.6999	0.4040	0.3782
109.1311		
361.4074		
202.0000		

$\text{Exact}_{(T_{cut}:T)} =$

TrackErr	QR_{mean}	QR_{std}
70.8644	0.3398	0.2603
97.0164		
329.8081		
189.4000		

$\text{Exact}_{(\text{Resh.})} =$

TrackErr	QR_{mean}	QR_{std}
69.1274	0.3491	0.2726
96.7015		
333.7506		
196.0000		

The Expert Agent performs very well in this quite more complex scenario, with a comparable performance to the Expert MPC Agent in phase 1 (as can be seen from Tracking Error and QR cost values in the first two tables above), and even slightly outperforming the Expert in phase 2 and 3 as can be seen from the QR mean and std. values (in the last two tables). It means that, using at each step all the available data and exploiting properly the strong feedback provided by corrective Expert advice on a more informative and exploratory system evolution trajectory, it manages to effectively approximate the desired control policy, getting very close to it when considering all the 1875 data-points at T_{cut}. At that moment, it achieves the definitive policy, which is solid and general, as maintains the same performance quality even with a different reference signal.

For the Subgradient Agents, as before, we provide a dedicated table for Tracking Errors and another for the QR histogram values (each column corresponds to a batch size). As there are three experimental phases, these tables occur three times:

$\text{TrackErr}^{Sub}_{(1:T_{cut})} =$

b	1	10	50	100	500
η_0	0.1	0.1	0.15	0.2	0.3
	143.0708	158.0417	151.0307	148.0287	137.2607
	173.7929	191.1613	193.4429	189.7824	194.3146
	340.9449	365.4101	361.1432	347.2082	366.3760
	206.0000	211.2000	216.6000	202.6000	206.0000

$\text{QR}^{Sub}_{(1:T_{cut})} =$

b	1	10	50	100	500
η	0.007	0.007	0.01	0.05	0.05
mean	0.5887	0.6568	0.6550	0.6122	0.6049
std.	0.4228	0.5112	0.4994	0.4577	0.4370

$\text{TrackErr}^{Sub}_{(T_{cut}:T)} =$

b	1	10	50	100	500
η_0	0.1	0.1	0.15	0.2	0.3
	139.3429	130.6066	122.6324	122.5295	110.2467
	194.7240	163.5221	150.3998	155.6899	151.5948
	366.7483	344.3096	345.6339	367.4752	334.8547
	195.8000	204.8000	209.0000	203.2000	199.0000

$\text{QR}^{Sub}_{(T_{cut}:T)} =$

b	1	10	50	100	500
η	0.007	0.007	0.01	0.05	0.05
mean	0.5783	0.5272	0.5037	0.4965	0.4370
std.	0.4146	0.3550	0.3515	0.3363	0.3101

$\text{TrackErr}^{Sub}_{(\text{Resh.})} =$

b	1	10	50	100	500
η_0	0.1	0.1	0.15	0.2	0.3
	139.0511	135.2987	121.4941	125.7185	107.7778
	200.9883	173.4219	155.9620	169.0034	148.1921
	362.1986	343.5729	346.6909	375.4641	332.0944
	201.8000	207.4000	211.2000	207.4000	203.4000

$\text{QR}^{Sub}_{(\text{Resh.})} =$

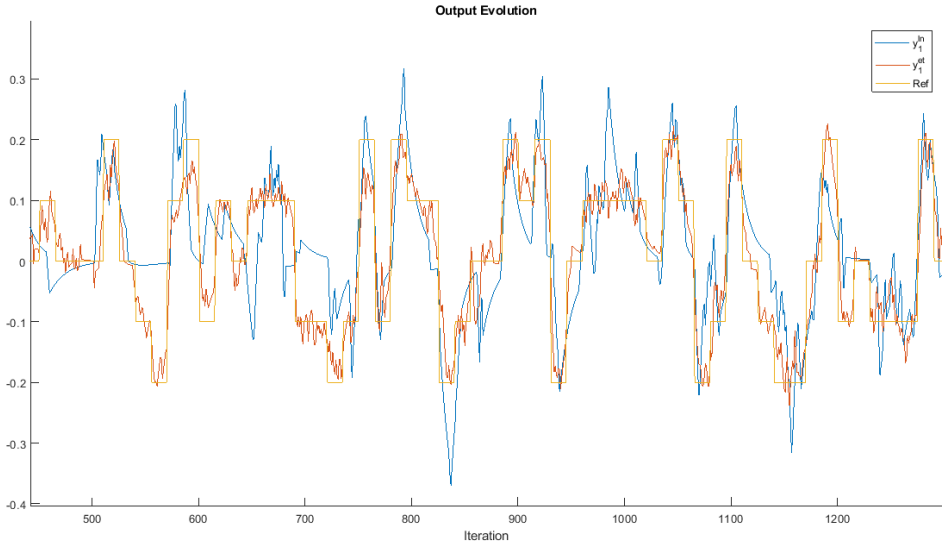
b	1	10	50	100	500
η	0.007	0.007	0.01	0.05	0.05
mean	0.6062	0.5226	0.5159	0.5320	0.4642
std.	0.4253	0.3950	0.5037	0.3570	0.3211

As before, performance increases with the batch size, but this time it is worse than the Expert, especially in phase 1, in which Tracking Errors and QR histogram values are significantly higher. In phase 2 and 3, they are lower and similar compared to each other meaning that the learned policy (although less precise than the Expert one) is consistent, even with a different reference signal. Such a worse performance is to be expected from the batch Subgradient Learner in this experiment, which consists of a more complicated task, limited learning time and very limited samples for each reference configuration. In learning iteratively, it is difficult for the learner to consistently descend the cost towards the target parameter and policy, as the incoming information is related to various constantly changing configurations. The learning process becomes noisier (in the evolution of the θ matrix) and does not manage to fully extrapolate the information underlying the data available for learning (data-points in the batch). That is especially the case for smaller batches.

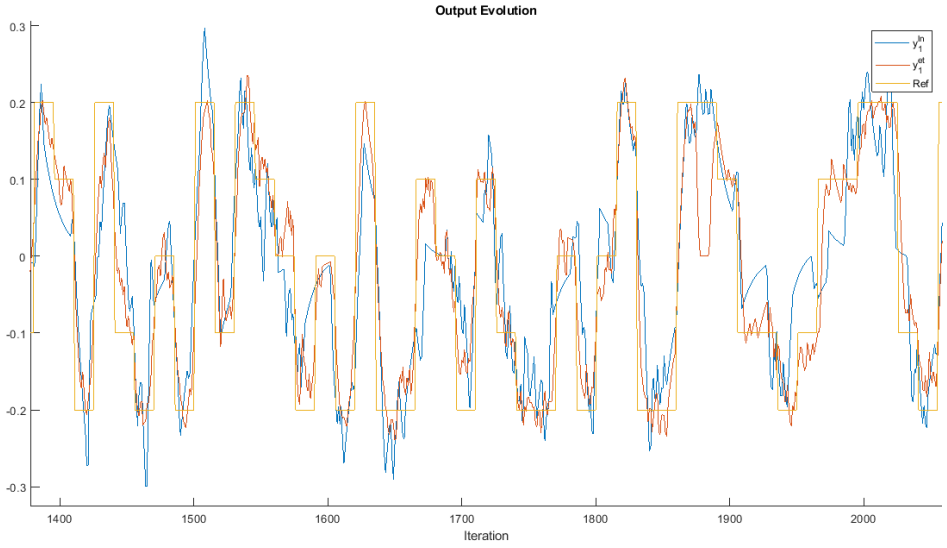
The best performance is achieved by the Agent using a batch size of 500, which yields the closest performance to the Expert, with QR mean 0.4642 and std. 0.3211, against Expert's mean 0.3575 and std. 0.2811. A way to improve the performance of our Subgradient Agents would be to have a longer training phase, for example repeating the same reference signal multiple n_{train} times, resulting in a new learning phase lasting $T^{\text{new}}_{\text{cut}} = n_{\text{train}} \cdot T_{\text{cut}} = n_{\text{train}} \cdot 1875$. This way the Learner has more time to actually complete the learning process and properly converge to a better parameter estimate, also having the chance to observe each reference configuration multiple times, each time with a more precise control action, closer to the de-

sired behaviour.

However, even if the numerical values of Error and QR cost appear to be significantly larger than the Expert ones (in the tables provided above), if we observe the actual plots of the outputs (Learner vs Expert, see Fig. 6-2), we can qualitatively conclude that our Agent performance is quite acceptable, especially when learning phase is completed. In general, it overshoots more than the Expert and fails to properly control the trajectories in certain specific reference configurations. That aspect is penalized quite heavily (as the cost penalty Q has high values), resulting in noticeably larger errors. In Fig. 6-2) we show a comparison of the y_1 trajectory between Batch Subgradient Learner Agent (batch size of 100) and Expert MPC Agent. More specifically, in Fig. 6-2a an excerpt of that trajectories for phase 1 of the experiment is shown (while learning is progressing, time span $1 : T_{\text{cut}}$), while in Fig. 6-2b for phase 2 (learning ended, time span $T_{\text{cut}} : T$). It is possible to observe how during the learning phase, control from our Learner Agent is less precise, especially in some specific reference configurations (for which evidently the right control actions were still not encoded and approximated in the parameter θ).



(a) Phase 1 (learning in progress) excerpt



(b) Phase 2 (learning completed) excerpt

Figure 6-2: y_1 trajectory, Batch Subgradient (b=100) vs Expert Agent

Chapter 7

Inverse Learning for classification/regression

It is possible to use the Inverse Learning Agent in classic ML problems such as classification and/or regression. In these problems, we have a training data-set, in which each data-point is constituted by certain input attributes (collected in the vector s) and some output target ones, which are the classes or variables to predict (collected in the vector u). The goal of the learning process is to achieve a hypothesis function $h(s)$ able to correctly predict the corresponding u vector, when tested on a validation data-set (new, unforeseen data). With our Inverse Learning Agent, we approach the problem exploiting the same mathematical formulation we had for the system control application. In fact, we can consider the target variables u as the decisions taken by an Expert, by minimizing a certain cost function $F^*(s, u)$.

It is an interesting approach to the problem, as we are personifying reality as a control Agent, treating certain observations as his control choices/decisions. For instance, in the task of diagnosing a sickness based on patients' medical records, we are assuming that there is an Expert Agent "Sickness" that, given the information contained in s, decides if the patient gets or not the disease (1 or 0). Or, alternatively, we could regard the real value of the target variable as the prediction made by an Expert Agent (like a doctor in the field) and try to learn to behave like him, achieving a parametrized decision policy. Actually, we are simply dealing with data gathered from real world situations and not from a control scenario that involves Agents. But we can model the problem as we wish (as long as it is mathematically coherent) and, in this case, treat "reality" as a prediction/control Agent. This way, as we are testing the Inverse Learning Agent on a different application and task, adapting it to the new situation, we can assess its performance on a more general level, not limited to the initial system control scenario for which we designed it and in which we tested it.

We gather several data-sets available on the Internet and experiment with our Inverse Learning approach. We compare our Agent with a baseline linear hypothesis class that we derive

with the least squares approach and with the results we could retrieve in related ML Literature. Some data-sets contain both a training and a validation set. For the others, from the unique data-set, we randomly select 70% of the points for the training one and the remaining 30% for the validation one. In this case, as training and validation depend on a random data sorting, the performance varies slightly between different program executions. That is especially the case for data-set smaller in size, as the whole learning/validation process depends on fewer data-points, whose specific selection plays a more important role. Anyhow, after running several experiments for the different cases, we observed that the change is in general not relevant and the results are quite consistent. For that reason, we provide only one result for each scenario, choosing one among those we achieved in a few runs of the program. From the related ML literature, we select the best result achieved by the authors, in order to have a performance comparison of our simpler Agent with more complicated ML approaches.

As performance indicators for simple classification (0/1 or negative/positive diagnosis), we compute Accuracy, Precision, Recall, Specificity and F1-score, which are widely used in the ML community. Let us indicate True Positives as TP, False Positives as FP, True Negatives as TN and False Negatives as FN. Then we have:

- **Accuracy** indicates the percentage of correct target u^{ln} predictions.
- **Precision** (or Positive Predictive Value) indicates how many of all the predicted positives are actually positive, computed as $\frac{\text{TP}}{\text{TP+FP}}$.
- **Recall** (or True Positive Rate) indicates what portion of positive data-samples the learner does predict as positives, and it is computed as $\frac{\text{TP}}{\text{TP+FN}}$
- **Specificity** (or True Negative Rate) indicates what portion of negative data-samples the learner predicts as negatives, and it is computed as $\frac{\text{TN}}{\text{TN+FP}}$
- **F1-Score** is computed as $\frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision+Recall}}$ and, combining Precision and Recall, provides a general performance indicator in one single numerical value. The higher is the score, inside the interval [0,1], the better performing is the predictor.

For multi-class classification problems, we exploit Accuracy, Mean Prediction Error and Mean Absolute Prediction Error as performance indicators. In these tasks, for the Linear Agent, we round the predicted output(s) to the nearest integer and floor the values to the desired set, which is $\{0,\ 1\}$ for simple classification (positive/negative diagnosis) or $\{1,\ ...,\ l\}$ for multi-class classification (with l different classes). For instance, if in a simple classification application the Linear Agent predicts $u = 2.1$, it will become $u = 1$, or if he predicts $u = 0.6$ it will become $u = 1$. The Inverse Learning Agent, by construction, directly outputs the desired value, without any additional processing.

As we are in a purely data-driven application, we can enrich our feature space as for instance $s_{R1} = [s; s^2; s^3]$ or $s_{R2} = [s_i \cdot s_j]$, with $i, j = 1, ..., n$ (full quadratic extension, taking into account all the possible inter-feature products). This is a very simple procedure, easily applicable (a few additional lines of code) and often used in ML applications. Regarding our Inverse Learning Agent, it can improve and usually does improve performance, increasing

accuracy. That is especially the case for the Linear Agent, which appears to benefit quite significantly from that feature extension. In fact, with the richer feature, he is able to capture non-linearities underlying the available data that he could not have captured by simply learning from the original data-set. His fit remains a simple linear hypothesis class, but the performance increases dramatically, especially in some cases (with accuracy in classification tasks surging from 50 or 60% to 80-90 % and F1-score surging from 0.5 or lower to 0.8-0.9). On the other hand, our Inverse Agent has already a stronger ground performance, with the classic data-set, in all the applications. However, he also benefits from the feature space extension and generally performs better than the Linear competitor (with better accuracy and F1-score). This shows the benefit of the quadratic parametrization (which intrinsically can take into account, to a certain extent, the non-linear aspects of the problem) and the strength of the "arg min" hypothesis function ($u = \arg\min_u F_\theta(s, u)$) even in this new and different application.

The achieved performance is generally comparable to the one retrieved in Literature and, in some cases, even better. However, in Literature they usually did not perform any feature space extension or pre-processing (although in some cases they do run algorithms for feature selection) and sometimes compute different performance indicators and dive more specifically into certain aspects of the application. We, on the other hand, dealt more with the numerical aspect. For almost all the problems, we achieve the Inverse Learning Agent through the Exact Program Solution (see Section 3-2). In case the problem is too large, as for the Facebook Comments Volume application, we use the Batch Subgradient Descent approach. If we use a certain feature extension approach, we mention it. Detailed information on Data-sets sources and authors can be found in the Appendix (see B). Additionally, for each application we provide a citation to the scientific papers that used the data-set as their case study.

7-1 Facebook comments volume prediction

In this application (see [SSK15]), the goal is to predict the number of comments for a certain Facebook page post over the next h hours, where h is given as one of the post attributes. There are 53 attributes for each data-point. Some of these are, for example, page popularity (number of likes, daily views, shares), number of posts published each weekday by the page, page category (firm, place, brand ...), post length, post publishing time and weekday, promotion status on the post, number of comments,likes and shares in several time spans (first 24 hours from publishing, last 24/48 hours, ...), some additional derived features, h (number of hours in the future over which we want to predict the volume of comments). Such a task could be exploited for example in various marketing applications for social media.

The interesting aspect is that we are dealing with actual data, gathered from Facebook. The authors, as they explain in [SSK15], used a software bot for crawling 2770 Facebook pages, storing information for 57000 posts, over a time span of several weeks. For each post, only the comments over the last 3 days (72 hours) are considered. From there, they processed the data, discarding all the instances with missing information and properly extracting the features. They divided the whole data-set in training and validation sub-sets (containing 80-20% of the original data-points, respectively), with a temporal split. It means that, w.r.t. the

temporal moment in which the split is performed, we have data-points from the past (which are used for training) and data-point from the future (which are used for testing), which is a good way to realistically assess the prediction ability of a predictor agent. They actually ran 5 parallel data-handling programs during the data-gathering phase, achieving 5 different data-set variants. Each variant v, extrapolates v data-points from each post observation. In fact, as said before, they monitored live the evolution of comments for many Facebook posts over 72 hours from the publication time. Each Variant v randomly selects v moments in time from that 72 hours time span, and stores the specific information (the 53 features) to the hard-drive, into the data-set. For instance, variant 1 contains 40949 data-points, while variant 2 contains 81312 ones. This way (in all the variants), there are different "sampling" times for each post, resulting in more general and informative data-sets, with variants 2-5 performing an oversampling. They then use several learning techniques, including Neural Networks and Decision trees on the five available data-sets and provide the achieved results. However, they use different performance indicators, so a direct performance comparison is not possible.

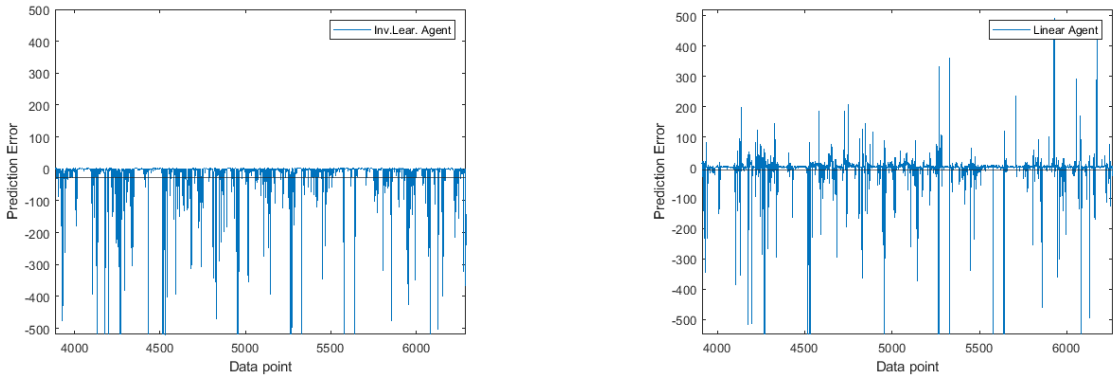
For our Inverse Learning Agent, we consider the task as a multi-class classification problem. In fact, the number of comments is integer and it takes values inside the range [1,2500], as we could observe from the training data-sets. For that reason, we consider each possible number of comments as a class to predict, which suits well our Integer Programming Algorithms. As the problem is quite large, containing many data-points and features, we perform learning with the Batch Subgradient Descent (batch size of 250) algorithm, for learning the best θ parametrization. As feature vector, we use the 53 attributes available for each data-point.

Numerical results We experiment with Variants 1-3 of the available data-sets, achieving similar and consistent results each time. It means that our Agents, in this application, do not benefit from the larger data-set size in variants 2 and 3. In fact, the additional data-samples do not really carry new information, but rather have redundant points, achieved with a sort of temporal oversampling. On the other hand, in the experiments carried out in [SSK15], the achieved results vary depending on algorithm and variant of the data-set. We present the results we achieved with Variant 1 of the Data-set in Table 7-1.

In general, our Agents manage to predict in a satisfactory way the number of comments, achieving a similar performance. However for same posts the prediction error is very large, even 2500 comments which is the maximum possible error. That probably happens because some data-points in the validation set (which is temporally subsequent w.r.t. the training one) may be significantly different from the ones observed during the training phase, and contain unforeseen information. For that reason, for these data-points, our Agents are not able to meaningfully generalize their prediction capabilities, as they never observed something similar in the training phase. Because of these cases of large miss-prediction, we additionally use median and 75-percentile of the Absolute Prediction Error as performance indicators, to gain some further information. The 75-percentile indicates the value under which 75% of the absolute error instances are contained.

First of all, it is interesting to notice that the Inv.Lear. prediction error is mainly one-sided.

In fact, most of the prediction error instances ($u^{ln} - u^*$) take negative values (with an average value of -28.2404) and for that reason our Agent has an pessimistic prediction behaviour. On the other hand, the Linear Agent has a more standard prediction behaviour, with both positive and negative values (although they tend to be negative, with an average of -7.6656). An excerpt of the prediction error plots can be observed in Fig. 7-1 (the black horizontal line is the average error).



(a) Inverse Learner **(b)** Linear Agent

Figure 7-1: Prediction Error

To really have a comparison between the two Agents and have a more precise performance indication, we use the Mean Absolute Error (MAE), which indicates the overall deviation of the prediction from the actual value (see Table in 7-1). Inverse Learner has a slightly larger value (28.9340) w.r.t. the Linear Agent (26.3221). However, it has a median Absolute Error of one, meaning that half of the Absolute Error values are below value 1. On the other hand the Linear Agent has a median of 5, meaning that half of the values are below 5 and the other half above 5. Similarly, with the indication of 75-percentile, we can observe that 75% of Inverse Learner Absolute Error instances are below value 12, while for the Linear Agent they are below value 16. Taking into account all that information, it is possible to conclude that Inverse Learner has a generally preciser prediction, with more instances of lower Absolute Prediction Error. That is easily observable in the histogram of the Absolute Error instances, in Fig. 7-2.

It is possible to observe how the Inverse Learner's histogram contains more instances of 0 and 1 values and, in general, a better shape than the Linear competitor. In fact, he also has a better accuracy (0.1111) than the Linear Agent (0.0571). However, as said before, he has some instances of very large miss-predictions, resulting in a MAE which is slightly larger than the Linear one.

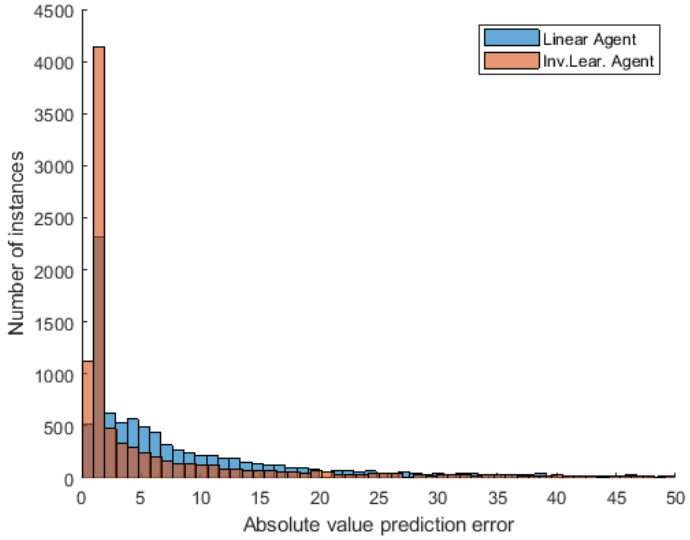


Figure 7-2: Absolute Error histogram

The following Table contains all the numerical results:

	Inv. Lear	Linear
Mean Error	-28.2404	-7.6656
Mean Absolute Error	28.9340	26.3221
Median Absolute Error	1.0000	5.0000
75-percentile Abs. Error	12.0000	16.0000
Accuracy	0.1111	0.0571

(7-1)

In conclusion, the Inverse Learner Agent performs better than the Linear one, with quite precise predictions (75% of the predictions are less than 12 comments away from the true value) allowing for some very erroneous predictions from time to time. However, that drawback is compensated by the advantage of knowing that most of his predictions are pessimistic (so we can expect always a larger number of comments than the predicted one) and by the fact that the majority of the Absolute Prediction Errors have lower values. There might be a reason behind the Inverse Learner better performance. In fact, as we are dealing with real social media data, the human behaviour comes into the frame. In fact, human behaviour is not always precisely predictable and in general it has underlying non-linear aspects. Here, for example, some specific post might receive comments because it triggers a certain response from a specific group of people, and in doing so, gain popularity and receive even more visibility and comments. Or, certain type of people might comment multiple times a post and get in arguments with other users, causing even more comments. From observing the available training data-set, our Agents try to extrapolate these patterns and fit their parameters to the available data-points. The Inverse Learner Agent, with its quadratic parametrization probably manages to capture more precisely the aforementioned non-linearities connected to human behaviour, achieving higher accuracy and generally lower prediction errors (at the price of some very large errors). The fact that the Inverse Learner Agent is learning a parametrized cost function, that defines its prediction policy, adds further complexity and might be the reason behind the one-sided error and pessimistic predictions.

7-2 Industrial machinery maintenance application

For this application, we use the "AI4I 2020 Predictive Maintenance Dataset" by S. Matzka and refer to [Mat20] for the results achieved by the author and more details on the data-set . It is a synthetic data-set, generated in such a way to reflect actual real-world maintenance data. It contains 10000 instances, each one with 6 attributes: Product ID, consisting of a letter H,M or L standing for High, Medium or Low machine quality, Air temperature [K], Process temperature [K], Rotational speed [rpm], Torque [Nm] and Tool wear [min]. When there is a failure, the target variable "Machine Failure" (MF) is set to 1, as well one of the following variables, which indicate the specific type of failure: tool wear failure (TWF), heat dissipation, failure (HDF), power failure (PWF), overstrain failure (OSF) and random failures (RNF). Depending on how one exploits the attributes and these failure variables, it is possible to perform both Fault Detection and/or Isolation tasks.

In [Mat20], the author achieves a fault detection Agent using SVM, NN and multiple bagged decision trees classifiers. Then, he approaches the Fault Isolation task with Explainable Artificial Intelligence (XAI), which aims at providing an user friendly explaination (when possible), for each specific fault. For instance, the Agent could provide as output ıPWF due to{torque $\leq$ 13Nm}ȷ.

7-2-1 Fault detection

In the fault detection task, we exploit the 6 aforementioned attributes as our Agent's feature (s), while the target variable (u) is the Machine Failure (MF). As the original data-set is highly unbalanced (only 339 MF points, out of 10000), our Agents deal with a quite challenging task. The best result is achieved when using the extended feature $s_r = [s; s^2; s^3; s^4]$. The most meaningful performance indicator here is the Recall (or True Positive Rate), as it indicates the ability to detect the Fault, and for both our Agents is quite low, with Inverse Learning Agent outperforming the Linear competitor (with Recall values 0.5426 and 0.2872 respectively). It means that the Inverse Learner did detect circa 54 faults out of 100, in the validation set. The comparison is pale with the result achieved in [Mat20], which has a Recall value of 0.8670. They used a bagged trees ensemble classifier (which combines multiple predictors, trained on different data-portions of the whole set, instead of one unique model achieved on a single training sub-set), optimized taking into account a very high cost for the False Negative occurrences and tested such a classifier with 5-fold cross-validation on the whole data-set (validate on 5 different randomly selected sub-sets of the original data-set). The additional fact that the faults in the data-set were generated using tree-like logical structures based on the attributes, makes their approach much more suited to the specific data-set, largely outperforming ours.

	Inv. Lear	Linear	Lit.Result
Precision	0.6296	0.9310	-
Recall	0.5426	0.2872	0.8670
Specificity	0.9897	0.9993	0.9870
F1-score	0.5829	0.4390	-
Accuracy	0.9757	0.9770	-

(7-2)

These results highlight the fact that our Inverse Learning Agent should be trained on more evenly distributed data-sets and for that reason it is not the best suited choice for Fault Detection tasks, where available data-sets are generally unbalanced (as faults occur rarely). However, in the following section, we test it in the Fault Isolation task, on the same data-set, achieving much better results.

7-2-2 Fault isolation

Now, let us assume that the Fault has been successfully detected and our task is to isolate it. In order to do so, we consider the six attributes and the MF variable as our feature s and wish to set to one the correct failure mode. Now, our target vector u has five entries, each corresponding to one failure mode. We train both our Agents with the full quadratic feature extension (s_{R2}) and achieve satisfactory results, with high recall an F1-score values. We compare the performance indicators separately for each target class and observe that both the Agents are comparable, with the Linear one having a worse isolation capability for the TWF class, with a Recall value of 0.53. As expected, the two Agents could not learn anything meaningful for the random failure (RNF), as it happens randomly and thus has no correlation with the feature space. We do not have comparable results from [Mat20], as for the Isolation task the author focuses mainly on the explanatory feature of his Agent and tests it on a small subset of the original data-set.

	Inv. Lear					Linear				
Failure mode	TWF	HDF	PWF	OSF	RNF	TWF	HDF	PWF	OSF	RNF
Precision	0.80	0.94	1.00	0.93	0.00	1.0	0.94	1.00	0.89	Nan
Recall	0.80	1.00	0.88	1.00	0.00	0.53	1.00	1.00	1.00	0.00
F1-score	0.80	0.97	0.94	0.96	NaN	0.70	0.97	1.0	0.94	Nan

(7-3)

7-3 Other results

We experiment with several other data-sets, from different fields and with different tasks. We provide a quick explanation for each of these, followed by the experimental results presented in a table and an analysis of thereof.

7-3-1 Classification

In these tasks, the target variable is binary (0,1).

Winsconsin breast cancer The goal is to predict if the cancer is benign (0) or malignant (1), with the knowledge of the attributes, which are breast cells properties, such as Uniformity of Cell Size, Uniformity of Cell Shape and Marginal Adhesion. We found two data-sets created by researches of University of Wisconsin Hospitals Madison (Wisconsin USA). In the first one (699 data-points), there are 10 attributes and they are integer valued, in the range 1-10 (the cell measurements were divided in 10 categories). In [WM90] the authors report a prediction accuracy of 96% (with a 67-33% split of the data-set for training and validation sub-sets, respectively). We train both our Agents on the full quadratically extended feature s_{R2}, achieving the following results, comparable to the ones reported in literature:

	Inv. Lear	Linear
Precision	0.9600	0.9091
Recall	0.9057	0.9524
Specificity	0.9873	0.9728
F1-score	0.9320	0.9302
Accuracy	0.9667	0.8714

(7-4)

In the second data-set (569 data-points), each point contains 30 real-valued features. These come from the same breast cell characteristics mentioned above (but now there is the actual numerical measurement, not a categorization), with additional information such as means, std. and worst mean (average of worst three values for each attribute) of the various measurements. In [SWM93], the authors report a 97% accuracy, with a 50-50 split of the data-set for training-validation. For our Agents, we use the classic feature s (no extension), achieving the following comparable results:

	Inv. Lear	Linear	Lit. Result
Precision	0.9714	0.9846	-
Recall	0.9444	0.8889	0.9000
Specificity	0.9798	0.9899	0.8600
F1-score	0.9577	0.9343	-
Accuracy	0.9649	0.9473	0.9700

(7-5)

Congress House votes In this data-set (436 data-points), originally provided in the 1984 United States Congressional Voting Records Database, each data-point represents a Congress member. The attributes are his votes (yes or no) on 16 key points, such as "anti satellite test ban", "educational spending", "religious groups in schools"... The target class is the member party affiliation (Republican or Democrat, encoded as 1 or 0 respectively). In [Sch87], the authors report a classification accuracy of 95%, which is the same we achieve with our Inv. Lear. Agent.

	Inv. Lear	Linear
Precision	0.9870	0.9730
Recall	0.9383	0.8889
Specificity	0.9796	0.9592
F1-score	0.9620	0.9290
Accuracy	0.9538	0.9154

(7-6)

Hepatitis In this data-set (155 data-points) the goal is to predict if a patient will die or live in the following year. The 19 features are indications on physical details of the patient, which are usually related to hepatitis (such as Liver Big, Fatigue, presence of certain chemical substances) as well as general information such as sex and age. In [DE83] a prediction accuracy of 80% is reported. For our Agents, using extended feature $s_{R1} = [s; s^2; s^3]$, we achieve a better performance (on the other hand, with the classic feature, we achieve a worse performance, with accuracy values around 60%). Here, as we are dealing with a small data-set, performance depends more on the random training-validation sub-sets split. However, performing various experiments, we always achieve an accuracy around 90%.

	Inv. Lear	Linear
Precision	0.8750	1.0000
Recall	0.7778	0.6667
Specificity	0.9730	1.0000
F1-score	0.8235	0.8000
Accuracy	0.9348	0.9348

(7-7)

Hypothyroid This data-set contains 3163 data-points, each with 23 attributes and one target class (hypothyroid diagnosis). The attributes are physical and medical properties of the patient, such as age, sex, if he/she is sick, had a tumor, is taking certain specific medications (binary valued), as well as some continuous valued measurements. We use the extended feature $s_{R1} = [s; s^2; s^3]$ for our Agents and the achieved results are shown in 7-8. Inv.Lear Agent performs better than the Linear competitor, with a better Recall (which is important in medical diagnosis tasks) and F1-score.

	Inv. Lear	Linear
Precision	0.8077	0.8571
Recall	0.8937	0.6383
Specificity	0.9889	0.9945
F1-score	0.8484	0.7317
Accuracy	0.9842	0.9768

(7-8)

Magic telescope In this data-set (19020 points), the aim is to predict if a certain image taken from a high energy gamma rays telescope contains a signal or simply background (1 or 0). The signal is being referred to as a gamma ray shower initiated by a gamma ray coming

from space, while background may be a gamma shower initiated inside the Earth atmosphere (and has different properties and energetic content). The 10 (continuously valued) attributes of each data-point are processed image properties, such as 10-log of sum of content of all pixels ("fSize"), ratio of sum of two highest pixels over "fSize", ratio of highest pixel over "fSize" and distance from highest pixel to center. In [BCG+04] the authors focus more specifically on the specific gamma ray signal detection application, perform different and more complex performance/validation tests, which are not directly comparable with our performance indicators.
We use the full quadratic extension of the feature space (s_{R2}) for our Agents (the classic feature would yield a slightly worse performance, with accuracy around 70%). In this application, the Linear Agent performs better than Inv. Lear. one, with higher values of Recall, F1-score and accuracy.

	Inv. Lear	Linear
Precision	0.8588	0.8522
Recall	0.8588	0.9464
Specificity	0.7300	0.6861
F1-score	0.8588	0.8968
Accuracy	0.8146	0.8570

(7-9)

7-3-2 Multi-class classification and regression

Wine quality The task is to determine the wine quality (class from 1 to 10) based on 10 physico-chemical properties, such as fixed acidity, pH, residual sugar and alcohol. There are two data-sets, one for red wines (1599 data-points) and one for white wines (4898 points). Both are unbalanced, as there are few examples for some of the target classes. In [CCA+09] authors report the achieved results, with SVM providing the best accuracy. They additionally compute the accuracy with tolerance ±1 (referred to as $tol \pm 1$), meaning that a prediction is considered correct even if differs by 1 from the actual value (for example if true wine quality is 8 and we predict 7, it is considered correct). We us the extended feature s_{R1} for both our Agents.

Red wine data-set We achieve results better than the ones reported in [CCA+09], with Inverse Learning and Linear Agents having comparable performance.

	Inv. Lear	Linear	Lit.Result
Accuracy	0.6479	0.6479	0.6240
$Accuracy_{tol\pm1}$	0.9812	0.9771	0.8900
Err_{mean}	-0.0083	-0.0292	-
$AbsErr_{mean}$	0.3708	0.3750	-

(7-10)

White wine data-set In this case, we achieve an overall accuracy lower than the one reported in [CCA+09]. However, when computing the accuracy with tolerance ±1, our Agents

outperform the one in Literature, as can be seen in the following Table. That coincides with the fact that the mean absolute prediction error is 0.51, meaning that for each prediction, our Agents are on average 0.5 away from the correct answer.

	Inv. Lear	Linear	Lit.Result
Accuracy	0.5449	0.5459	0.6460
$Accuracy_{tol\pm1}$	0.9544	0.9523	0.8680
Err_{mean}	0.0184	0.0177	-
$AbsErr_{mean}$	0.5085	0.5078	-

(7-11)

Something interesting to notice, also in the following applications, is that when our Inverse Learner predicts a wrong class, it is not numerically distant from the true one (mean absolute error is low). That happens because he is numerically optimizing the learned cost $F_\theta(s, u)$ in order to perform the prediction, providing the class number that approximates better (according to his policy) the real one. So our Learner is most suited for multi-class problems where the classes are ordered. However, if we are dealing with several classes that are not ordered, we loose this benefit and only the prediction accuracy is a meaningful performance indicator (as for example in the following applications "arrhythmia" and "dermatology").

Arrhythmia The aim of this data-set (which contains 452 data-points) is to predict one of 16 possible classes of arrhythmia, with the provided knowledge of 279 attributes. These are patients' information (sex, age, weight, height ...) and many hearth related medical measurements, such as heartbeat rate, blood pressure and similar. Regarding the target variable, class 1 is the normal condition, 2 to 15 are several arrhythmic conditions and class 16 contains all the other unspecified situations. The data-set is quite unbalanced, with very few (or none) examples for some of these classes. We use the extended feature $s_{R1} = [s; s^2; s^3]$ for our Agents (the classic feature yields similar, slightly worse, performance), achieving the results shown in 7-12. Inverse Learning Agent outperforms the Linear one, with an accuracy value of 74.26%, comparable to the results achieved in [GK14] (78% accuracy). Actually they manage to get an uccuracy up to 86% performing intelligent feature and classes selection and exploiting SVM.

	Inv. Lear	Linear
Accuracy	0.7426	0.6838
Err_{mean}	0.4926	1.7426
$AbsErr_{mean}$	2.2132	3.0662

(7-12)

Dermatology The goal in this application is to predict one of six possible dermatological pathologies based on clinical data from the patient. There are 34 clinical attributes, many involving skin characteristics and analysis. The data-set contains only 366 data-points and is uneven, so the results are not satisfactory in this case, with the Inverse Agent having a prediction accuracy of only 5.92% and Linear Agent of 2.63% (both exploited the extended feature s_{R1}).

	Inv. Lear	Linear
Accuracy	0.0592	0.0263
Err_{mean}	-0.1704	-0.2012
$AbsErr_{mean}$	1.9138	1.8862

(7-13)

Winsconsin breast cancer - prognostic This is a quite challenging data-set and task, created by the same authors of [WM90] and [SWM93] and containing the same attributes of [SWM93] (30 real valued clinical measurements of breast cells properties). In fact, the goal is to predict two target variables, one binary ($u1$) that indicates if the cancer is recurrent/non-recurrent, and one integer valued ($u2$) that indicates time (in days). If the cancer is recurrent, time indicates for how many days it will recur. Otherwise, if the cancer is non-recurrent, time indicates the number of predicted disease-free days. Due to the facts that the two target variables are strongly correlated and that there are only 198 available data-points, the problem is quite complex.
We use the full quadratic feature extension s_{R2} for our Agents, and achieve comparable results. The Inv. Lear. Agent has a slightly better performance, with higher accuracy and F1-score for u1, as well as a slightly higher accuracy and lower mean absolute error for u2.

		Inv. Lear	Linear
u1	Precision	0.6875	0.5714
	Recall	0.7333	0.8000
	Specificity	0.8864	0.7955
	F1-score	0.7097	0.6667
	Accuracy	0.8475	0.7966
u2	Accuracy	0.6441	0.6271
	Err_{mean}	5.2034	1.5424
	$AbsErr_{mean}$	20.136	21.9831

(7-14)

Boston Housing prices The goal is to predict housing prices in the Boston suburbs. The data-set (created in 1978 by Harrison, D. and Rubinfeld, D.L. and used in [HJR78]) contains 506 housing instances, each one with 13 attributes (such as, for example, weighted distances to five Boston employment centres, index of accessibility to radial highways, per capita crime rate by town and number of rooms) and one target class (price). The price is encoded with decimal numbers (the unity represents one thousand dollars), in a certain range, so we can perform a discretization and treat the regression problem as a multi-class classification problem. Each admitted price value is treated as a class. Obviously, we expect a low accuracy, as there are many classes and it is very difficult for any learning program to predict with total precision the actual price. But the mean absolute error is a good indication of the goodness of prediction. The Inv. Lear. Agent prediction is on average 1.91 thousands of dollars away from the real price, while the Linear Agent performs slightly better, with 1.88 (for both the Agents, we us the full quadratic extension of the feature, s_{R2}). However, the Inv. Lear. Agent has a better accuracy (5.92% against 2.63% of the Linear competitor). Taking into account these observations, we can conclude that he manages more often to predict the actual

price than the Linear Agent, but in general he can yield some more erroneous predictions for certain data-points (resulting in that slightly larger mean absolute prediction error).

	Inv. Lear	Linear
Accuracy	0.0592	0.0263
Err_{mean}	-0.1704	-0.2012
$AbsErr_{mean}$	1.9138	1.8862

(7-15)

Chapter 8

Conclusion

The Inverse Learning algorithm has proven to be a solid tool for control/prediction applications. In the control setting, extensive testing for different tasks with the complex Heavy Oil Fractionator system has assessed the performance of the Inverse Learning Agent, which is comparable to the Expert (MPC) one. It has the advantage of capturing, with a certain degree of approximation, the Expert control policy in a much simpler and convenient formulation, that has significantly lower computation times, especially for the integer control space we are dealing with (as we exploit a combinatorial approach for solving optimization programs). We achieved these results for both offline and online learning scenarios. In the latter, the task is more complex, as our Agent must directly control the system while, at the same time, learning from Expert corrective advice. Additionally, the Agent has a limited time to do so, as at time step T_{cut} the learning process terminates and it keeps acting on the system with the last achieved parameter θ approximation. However, as we discussed, in this scenario he receives richer information for the learning process, which allows him to relatively quickly learn a solid policy, even in such a challenging task. As he is learning a parametrized cost over the feature space $\mathbf{S}$, it has the ability to generalize the learned policy also over system configurations that he did not observe directly. However, it needs a reasonable degree of exploration during the learning process. In fact, if we train it in one region of the feature space and test it on another region, in which the system follows a different behaviour, the control it applies will be imprecise. We proposed three different learning algorithms ("Exact" program, Subgradient Descent and MP), compared their learning processes and tested their performance, suggesting how to choose the most appropriate one for each specific application.

Regarding the classification task, our Agent has often achieved comparable performance to more complicated approaches (and tailored to the specific application) exploited in literature. In almost all the cases, it outperformed the Linear Agent competitor (with enriched feature, so it actually can interact with nonlinear information available in the set and it is more than a simple linear fit). The use of the Inverse Learning Agent in this application is very interesting for the way the problem is modeled. In fact, as the framework is the same one used for the control application, we are assuming the presence of an Expert Agent, which minimizes a

cost and provides the values in the target variable(s) u. For that reason, we are regarding nature/reality as an Agent that is acting in a certain unknown optimal way, and we are trying to approximate that fictitious and unknown control/prediction law through our parametrization. Considering that aspect, the successful results we achieve with a variety of data-sets become more remarkable and surprising. However, it is to notice that it performs best for evenly distributed training data-sets (which matches well the discussion on feature space exploration, in the control application) and it is not the best suited choice for problematic data-sets and/or too specific tasks (such as fault detection). Also, for multi-class classification problems, it performs well if the classes are ordered. In fact, if that is the case, even if the learned decision policy is not very precise, it will output predictions that are numerically close to the true value (see Wine Quality data-sets, where the classes are the ordered quality score $1 - 10$). On the other hand, if we simply have a mix of classes (categories) in which we need to sort the observed data-points, then our algorithm can perform much worse. In fact, even if the (approximated) prediction is numerically close to the true one, that is irrelevant as we need a precise class prediction (that is the case, for example, in the Dermatology data-set, for which performance is not satisfactory).

To conclude, we provide some indications for possible future research in the field:

- Extrapolate efficient feature and learning rate directly from system/data knowledge
- Test Inv. Lear. Agent on other systems, with a richer set of available integer inputs (for example $u \in \{-u_{min}, ..., -1, 0, +1, ..., u_{max}\}$)
- Add disturbances acting on the system and incorporate them in the learning process
- Test with unstable systems
- Train Agent through purely numerical simulations and then apply directly on the physical system
- Formal analysis of convergence rates and performance guarantees
- Explore accelerated variants of Subgradient Descent
- Continue research in classification applications
- Mix-Integer Learning Agent

Appendix A

Appendix

A-1 Bilinear reformulation

It is possible to achieve a more compact formulation of our optimization program, which takes the form of a smooth saddle point problem. This formulation is needed for implementing the Mirror Prox algorithm ([Nem04]), that is explained in the following section. Recalling the mathematical framework and notation introduced in Chapter3, having a training data-set $\{\hat{s}_t, \hat{u}_t\}_{t=1,\dots,T}$ (previously we referred to $\hat{u}_t$ as $\hat{u}_t^{et}$, but here we keep a more readable notation) we have that our problem is:

$$min_{\theta\in\Theta} \sum_{t=1}^{T} l^{sub}(\hat{s}_t, \hat{u}_t, u^{ln}) \tag{A-1}$$

$$\text{with} \quad l^{sub}(\hat{s}, \hat{u}, u) = max_{u\in\mathbf{U}} \ \{ F_\theta(\hat{s}, \hat{u}) - F_\theta(\hat{s}, u) \} \tag{A-2}$$

$$\text{and} \quad \theta = \begin{bmatrix} 0 & \theta_{su} \\ \theta'_{su} & \theta_{uu} \end{bmatrix} \tag{A-3}$$

If we plug the parameter θ definition in the loss function, we achieve:

$$l^{sub}(\hat{s}, \hat{u}, u) = max_{u\in\mathbf{U}} \ \{ 2\hat{s}_t^\top \theta_{su}(\hat{u}_t - u_t) + \hat{u}_t^\top \theta_{uu}\hat{u}_t - u_t^\top \theta_{uu} u_t \} \tag{A-4}$$

$$\tag{A-5}$$

The function is linear in θ, but it is involved in quadratic matrix products with feature and control input vectors. In order to express explicitly that linearity, we exploit the Kronecker Identity ($\otimes$ denotes the Kronecker product and vec() the vectorization operator):

$$\text{vec}(A \cdot \Theta \cdot B) = (B' \otimes A) \cdot \text{vec}(\Theta) \tag{A-6}$$

We achieve:

$$l^{sub}(\hat{s},\hat{u},u) \;=\; max_{u\in\mathbf{U}} \;\{\; 2((\hat{u}_t-u_t)\otimes\hat{s}_t)^\top \mathrm{vec}(\theta_{su}) + (\hat{u}_t\otimes\hat{u}_t+u_t\otimes u_t)^\top \mathrm{vec}(\theta_{uu}) \;\} \tag{A-7}$$

$$= \; max_{u\in\mathbf{U}} \;\left\{ \begin{bmatrix} 2(\hat{u}_t-u_t)\otimes\hat{s}_t \\ \hat{u}_t\otimes\hat{u}_t+u_t\otimes u_t \end{bmatrix}^\top \begin{bmatrix} \mathrm{vec}(\theta_{su}) \\ \mathrm{vec}(\theta_{uu}) \end{bmatrix} \right\} \tag{A-8}$$

$$= \; max_{u\in\mathbf{U}} \;\{\; a_t(\hat{s}_t,\hat{u}_t,u_t)^\top \theta_{\mathrm{vec}} \;\} \tag{A-9}$$

At this point, it is possible to reformulate the maximization program involved in the achieved formulation, through an additional simplex vector variable $y \in \Delta_M$ (where Δ_M denotes the simplex vector space of dimension M). We recall that the sum over the entries of a simplex vector equals one. In order to exploit it for solving the maximization program, we can consider all the possible input vector combinations M (recalling that we are in the integer scenario and $u \in \mathbf{U}$) and, ideally, set to one the entry in the simplex corresponding to the best input and to zero the remaining ones. As we are optimizing over the data-set $\{\hat{s}_t,\hat{u}_t\}_{t=1,\dots,T}$, we must perform this operation for all the T data-instances. For that reason, we have T simplex variables and ultimately $y \in \Delta_M^T \;=\; \{\Delta_M \times \cdots \times \Delta_M\}$. The mathematical derivation is the following:

$$\begin{aligned}
l^{sub}(\hat{s},\hat{u},u) \;&= \sum_{t=1}^{T} \max_{u_t\in\mathbb{U}} \; a_t(\hat{s}_t,\hat{u}_t,u_t)^\top \theta_{\mathrm{vec}} \\
&= \sum_{t=1}^{T} \max \Big\{ a_t(\hat{s}_t,\hat{u}_t,u(1))^\top \theta_{\mathrm{vec}}, \dots, a_t(\hat{s}_t,\hat{u}_t,u(M))^\top \theta_{\mathrm{vec}} \Big\} \\
&= \sum_{t=1}^{T} \max_{y_t\in\Delta_M} \begin{bmatrix} a_t(\hat{s}_t,\hat{u}_t,u(1))^\top \theta_{\mathrm{vec}} \\ \vdots \\ a_t(\hat{s}_t,\hat{u}_t,u(M))^\top \theta_{\mathrm{vec}} \end{bmatrix}^\top y_t \\
&= \sum_{t=1}^{T} \max_{y_t\in\Delta_M} \theta_{\mathrm{vec}}^\top \Big[a_t(\hat{s}_t,\hat{u}_t,u(1)) \cdots a_t(\hat{s}_t,\hat{u}_t,u(M)) \Big] y_t \\
&= \sum_{t=1}^{T} \max_{y_t\in\Delta_M} \theta_{\mathrm{vec}}^\top A_t y_t \\
&= \max_{y\in\{\Delta_M\times\cdots\times\Delta_M\}} \theta_{\mathrm{vec}}^\top \Big[A_1 \cdots A_T \Big] y \\
&= \max_{y\in\{\Delta_M\times\cdots\times\Delta_M\}} \theta_{\mathrm{vec}}^\top A y,
\end{aligned}$$

with $A_t := [a_t(\hat{s}_t,\hat{u}_t,u(1)) \cdots a_t(\hat{s}_t,\hat{u}_t,u(M))]$, $A := [A_1 \cdots A_T]$ and $y := [y_1^\top \cdots y_T^\top]^\top$. To make a concrete example, let us consider our case study (Oil Fractionator system) in the single reference control task, with a data-set containing $T = 1000$ data-points. We have that size of s is 32 (n) and size of u is 3 (m). For that reason, size of $vec(\theta_{su})$ is $nm = 96$

and size of $vec(\theta_{uu})$ is $mm = 9$. Combining the two, we achieve that the size of $vec(\theta)$ is $nn+mm = 105$. As each input can take the values $\{-1, 0, +1\}$, we have 27 total possible input vector combinations. Then, we have that each a_t has size 105×1, A_t has size 105×27, each y_t has size 27×1. Combining all the available information, we get A with size 105×27000 and y with size 27000×1. Recalling that the problem is to find the parameter θ that minimizes the loss function (see Eq. A-1), we get:

$$\arg\min_{\theta\in\Theta} \max_{y\in\Delta_M^T} \theta_{\text{vec}}^\top A y \tag{A-10}$$

It is interesting to notice that in the offline learning scenario, the A matrix is pre-computed (as all the required $\{\hat{s}_t, \hat{u}_t\}_{t=1,\dots,T}$ points are available) and then exploited during the learning process, making it more efficient.

More importantly, this is a bilinear formulation of the problem (as the objective is linear in both the optimization variables θ and y). Additionally, as we are minimizing over θ and maximizing over y, we are dealing with a saddle-point problem. It is now possible to use the Mirror Prox algorithm.

A-2 MP - Mirror Prox algorithm

Starting from the mathematical reformulation achieved in Section A-1 and defining $\gamma(\theta, y) := \theta_{\text{vec}}^\top A y$ (where $\gamma(\theta, y)$ is the sub-optimality loss, in the compact form that considers all the possible input combinations and available training data-points), the program in (A-10) is

$$\min_{\theta\in\Theta} \max_{y\in\Delta_M^T} \gamma(\theta, y). \tag{A-11}$$

The Mirror Prox algorithm ([Nem04]) optimizes iteratively over the variables θ and y and their support variables λ and z, respectively. Se we have that $\lambda, \theta \in \Theta$ and $z, y \in \triangle_M^T$. The iterative updates follow the Subgradient Descent/Ascent principle (for minimizing λ and θ and maximizing y and z) and for that reason need two learning rates η_a and η_b. It uses a mapping Φ to a Dual Space, for updating the variables z and y. The mapping, w.r.t. a generic variable x, is $\Phi(x) = x\ \log(x)$ and its gradient is $\nabla_x \Phi(x) = \log(x) + 1$. With that mapping, the iterative update for a generic vector variable x would be achieved as follows:

$$\begin{aligned} \nabla\Phi(x_{t+1}) &= \nabla\Phi(x_t) + \eta \nabla_x l^{sub} \\ \log(x_{t+1}) &= \log(x_t) + \eta \nabla_x l^{sub} \\ x_{t+1} &= x_t\ e^{\eta \nabla_x l^{sub}} \end{aligned}$$

Ultimately, we end up with an exponentiated update and the mapping (and inverse mapping) to the dual space is actually implicit. We will apply that update on our specific problem and

variables.

As mentioned before, we need to define two gradients, for the two optimization variables θ and y (and their support variables λ and z). Remembering that the loss function has now the structure of $\gamma(\theta, y) := \theta_{\text{vec}}^{\top} A y$, the gradients are trivially:

$$\nabla_{\theta}\gamma(\theta, y) = Ay, \quad \nabla_{y}\gamma(\theta, y) = A^{\top}\theta_{\text{vec}}.$$

The algorithm updates the four variables at each learning iteration in an interconnected fashion, managing this way to improve the convergence efficacy. The **algorithm** is ($\odot$ indicates the element-wise product and exp() indicates the element-wise exponential function):

$$\theta_t = \Pi_{\Theta}\left(\lambda_t - \eta_a \nabla_{\theta}\gamma(\lambda_t, z_t)\right) \tag{A-12}$$

$$y_t = \Pi_{\Delta_M^T}\left(z_t \odot \exp(\eta_b \nabla_{y}\gamma(\lambda_t, z_t))\right) \tag{A-13}$$

$$\lambda_{t+1} = \Pi_{\Theta}\left(\lambda_t - \eta_a \nabla_{\theta}\gamma(\theta_t, y_t)\right) \tag{A-14}$$

$$z_{t+1} = \Pi_{\Delta_M^T}\left(z_t \odot \exp(\eta_b \nabla_{y}\gamma(\theta_t, y_t))\right) \tag{A-15}$$

$$\tag{A-16}$$

with Π_{Θ} being the projection matrix that ensures the constraint on the eigenvalues of θ_{uu} is respected and $\Pi_{\Delta_M^T}$ is a projection operator on the simplex space (as y and z must be simplex variables). It simply performs a vector normalization (all entries must sum up to one), as follows:

$$\Pi_{\Delta_M^T}(y) = \Pi_{\Delta_M^T}\left(\begin{bmatrix} y_1 \\ \vdots \\ y_T \end{bmatrix}\right) := \begin{bmatrix} y_1/\|y_1\|_1 \\ \vdots \\ y_T/\|y_T\|_1 \end{bmatrix}.$$

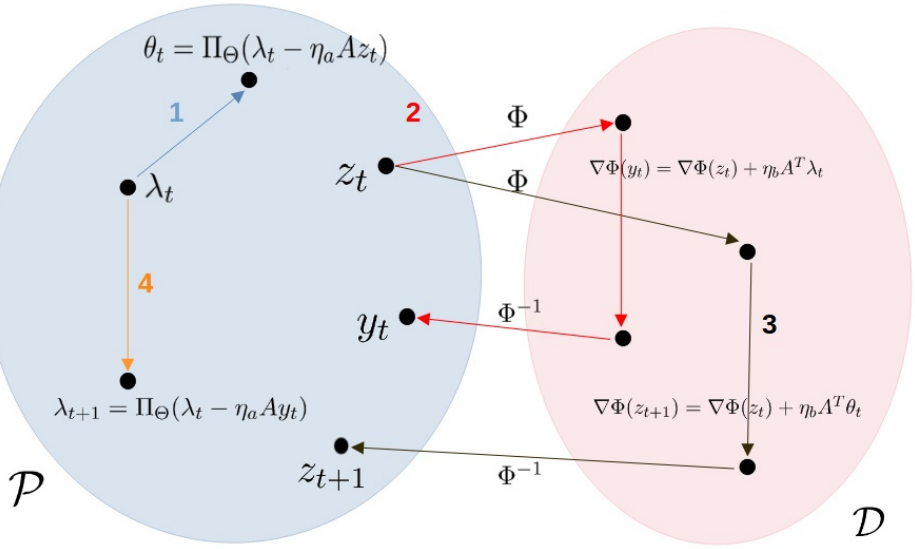
In Fig. A-1 it is possible to observe a visualization of one learning iteration of the algorithm (for all the four variables involved), with the mapping to the dual space.

Following the suggestion in literature, the parameter estimate that we monitor and use is the averaged θ, over the learning iterations k. That is done in order to ensure a smoother evolution of the parameter, as in a saddle-point problem some variables are ascending and others descending, causing an overall noisier and oscillatory parameter evolution.

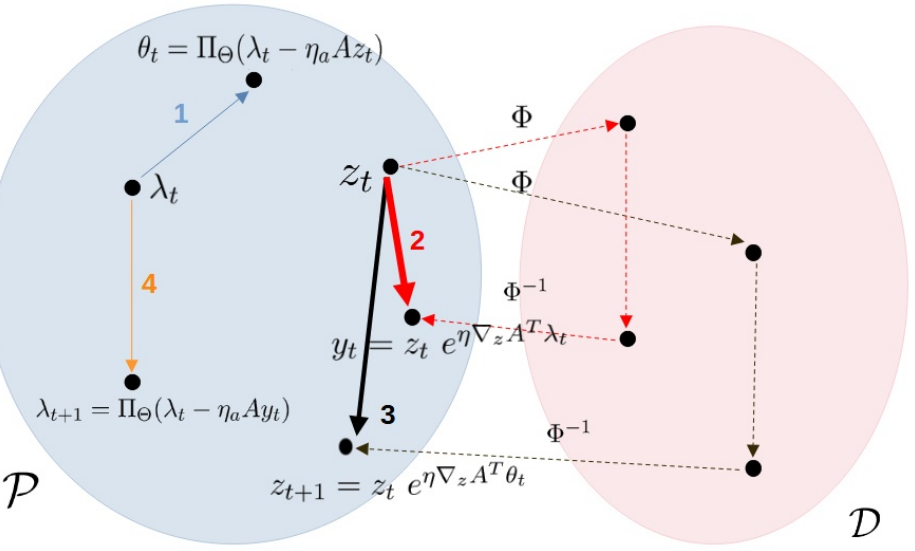
$$\bar{\theta} = \frac{1}{k}\sum_{t=1}^{k}\theta_t \tag{A-17}$$

The algorithm is initialized with the following generic settings: $\lambda_{su} = 0$, $\lambda_{uu} = I$, $\lambda_1 = [\text{vec}(\lambda_{su})^{\top}\text{vec}(\lambda_{uu})^{\top}]^{\top}$ and $z_1 = \frac{1}{M \cdot T}[1 \cdots 1]^{\top}$ (the simplex vectors are initialized in the middle of the simplex space, equidistant from the vertices).

As reported in [Nem04], Mirror Prox achieves a convergence rate $O(1/k)$ (with k number of learning iterations), faster than Subgradient Descent (for which convergence rate is $O(1/\sqrt{k})$. It manages to do so with the four interconnected updates of main and support variables and of the exploitation of the dual space through the mapping ϕ. That mapping deforms the original space and the update performed in the dual space results to be more significant, when considered again in the primal space. However, that happens with the correct learning rate tuning (of both η_a and η_b), which in this case appears to be more difficult (and also more important) than for the other algorithms. In Section 5-3, we explore empirically and compare the various algorithms learning processes, showing that MP converges in less iterations (however each iteration is slower).



(a) Explicit Dual Space Mapping and Update



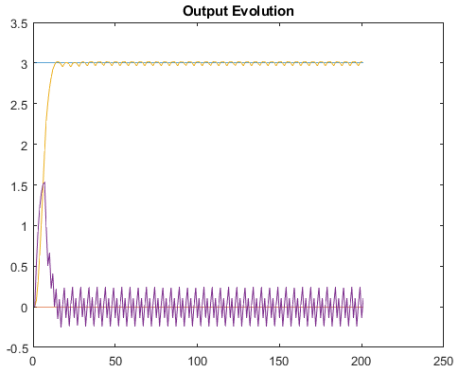
(b) Actual Update, Implicit Mapping

Figure A-1: MP Algorithm visualization

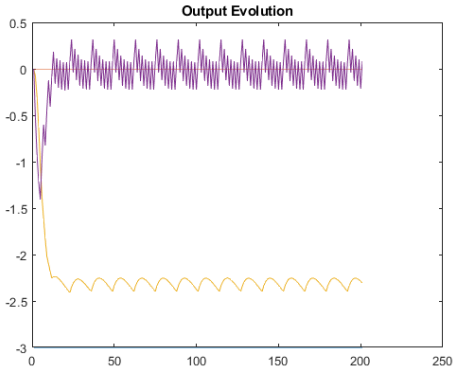
A-3 Additional result on exploration needed during learning phase

Here we show with a very simple experiment the exploration factor needed by the Inverse Learner Agent for achieving a solid general policy. We perform offline learning and Out-of-Sample testing (see Chapter 5 for details) for a simple Damped Mass-Spring system. We assume full-state measurement (x_1 position and x_2 velocity of the mass) and want to perform reference tracking of the mass position. The training data-set contains points from the trajectory in which the Expert is controlling the mass to the constant reference of 3 meters ($x_r = [3; 0]$). Offline learning is performed with the Subgradient Descent algorithm. We then test the learned policy with Out-of-Sample testing.

When we test the Learner with a positive position reference (and also the original $x_r = [3; 0]$ contained in the training data) the control is successful and comparable to the Expert. If we test it on a negative position reference, for instance $x_r = [-3; 0]$, we observe that the Learner is not able to perform an accurate reference tracking, positioning the Mass at -2.3 meters circa. This means that the learner cannot generalize over the whole state-space, from limited training information. In fact, the training data was achieved in a state-space region in which the spring is applying a negative force (as the mass position is positive w.r.t. the rest position). Then, when the reference is set in the region where the spring applies a positive force (as the mass position is negative), the learned policy considers that the spring will always apply a positive force (as that is what he observed in the training phase) and thus fails to track the reference properly.



(a) Tracking Position Reference 3



(b) Tracking Position Reference -3: Failure

Figure A-2: Learner controls with learned Θ: Yellow signal is position, Magenta is velocity

That gave us some insight on the fact that exploration is needed in order to achieve a solid controller, over the whole state-space (Learner is not able to generalize too well from simply observing/acting on a limited region of the state-space). In fact, if we learn from a reference signal containing values in different functioning regions of the system, then the learned policy will be able to control the system in these regions. For instance, if we learn from a reference

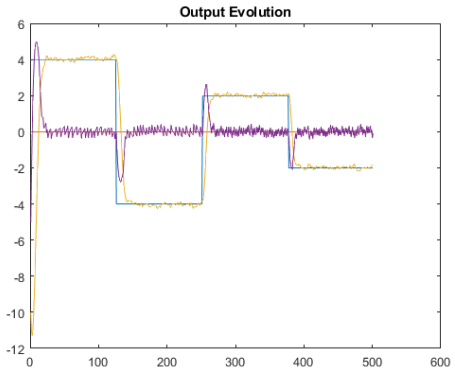
signal with different reference values (which lie in both the functioning regions of the system):

$$x_r = \begin{bmatrix} 4 & -4 & 2 & -2 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

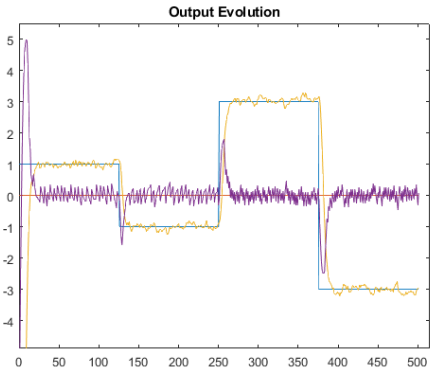
then the learned policy will be able to control the system and successfully track that reference, as well as a different one, for instance:

$$x_r = \begin{bmatrix} 1 & -1 & 3 & -3 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

It is possible to observe the result in the following figures (in the experiment system is initialized at an initial state x_0).



(a) Control on the reference signal used for training



(b) Control on different reference signal

Figure A-3: Learner controls with learned Θ: Yellow signal is position, Magenta is velocity.

So in general, the Inverse Learning Agent generalizes the policy even over unexplored feature-space regions, but on the basis of the knowledge that was available to him in the learning phase. For that reason, if we use it in regions where the behaviour of the system is different (here for example in the negative position area) w.r.t. the ones in which he was trained, he will "inductively" generalize and apply his control choices. However, that might result in a control action that is imprecise (like in this Mass-Spring example) or even fail for more complex systems.

Appendix B

Data-sets info

All the Data-Set can be found in two open online repositories:

1. https://archive.ics.uci.edu/ml/machine-learning-databases/
2. https://archive.ics.uci.edu/ml/datasets.php?format=&task=reg&att=&area=&numAtt=&numIns=&type=&sort=nameUp&view=table

There, for each Data-set, it is possible to find additional information, as well as more literature in which these Data-sets were used (other than the one we cited).

Facebook Comments Volume Source: Kamaljot Singh, Assistant Professor, Lovely Professional University, Jalandhar. Kamaljotsingh2009 '@' gmail.com
Data-set was analyzed and exploited by the creator in [SSK15].
Available at: https://archive.ics.uci.edu/ml/datasets/Facebook+Comment+Volume+Dataset.

Industrial Maintenance - AI4I 2020 Data-set Source: Stephan Matzka, School of Engineering - Technology and Life, Hochschule fur Technik und Wirtschaft Berlin, 12459 Berlin, Germany, stephan.matzka '@' htw-berlin.de
Dataset- S. Matzka, "AI4I 2020 Predictive Maintenance Dataset", submitted to UCI Machine Learning Repository, 2020. Used by the author in [Mat20].
Available at https://archive.ics.uci.edu/ml/datasets/AI4I+2020+Predictive+Maintenance+Dataset

Winsconsin Breast Cancer Data-sets available at:
https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/.

1) First Data-set (integer valued attributes) This breast cancer databases was obtained from the University of Wisconsin Hospitals, Madison from Dr. William H. Wolberg. Sources:
– Dr. WIlliam H. Wolberg (physician) University of Wisconsin Hospitals Madison, Wisconsin USA
– Donor: Olvi Mangasarian (mangasarian@cs.wisc.edu) Received by David W. Aha (aha@cs.jhu.edu)
– Date: 15 July 1992
We refer to [WM90] from the same authors.

2-3) Diagnostic and Prognostic Data-sets (real-valued measurements) 1995 Source:
Dr. William H. Wolberg, General Surgery Dept., University of Wisconsin, Clinical Sciences Center, Madison, WI 53792 wolberg@eagle.surgery.wisc.edu
W. Nick Street, Computer Sciences Dept., University of Wisconsin, 1210 West Dayton St., Madison, WI 53706 street@cs.wisc.edu 608-262-6619
Olvi L. Mangasarian, Computer Sciences Dept., University of Wisconsin, 1210 West Dayton St., Madison, WI 53706 olvi@cs.wisc.edu
Used by the authors in [SWM93].

Congress House Votes Available at:
https://archive.ics.uci.edu/ml/machine-learning-databases/voting-records/

Source: Congressional Quarterly Almanac, 98th Congress, 2nd session 1984, Volume XL: Congressional Quarterly Inc. Washington, D.C., 1985.
Donor: Jeff Schlimmer (Jeffrey.Schlimmer@a.gp.cs.cmu.edu)
Date: 27 April 1987
We refer to [Sch87] for the reported results.

Hepatitis Available at:
https://archive.ics.uci.edu/ml/machine-learning-databases/hepatitis/

Donor: G.Gong (Carnegie-Mellon University) via Bojan Cestnik Jozef Stefan Institute Jamova 39 61000 Ljubljana. Date: November, 1988.
We refer to [DE83] for results in literature.

Hypothyroid This data-set was found on the aforementioned open repository, no comprehensive information on data-set, its sources and its use was provided.

Magic Telescope MAGIC gamma telescope data 2004, Sources:
Original owner of the database:
R. K. Bock Major Atmospheric Gamma Imaging Cherenkov Telescope project (MAGIC) http://wwwmagic.mppmu.mpg.de rkb@mail.cern.ch

Donor: P. Savicky Institute of Computer Science, AS of CR Czech Republic savicky@cs.cas.cz
Date received: May 2007.
We refer to [BCG+04] for past usage and reported results.

Wine Quality Data-sets The two data-sets were created by Paulo Cortez (Univ. Minho), Antonio Cerdeira, Fernando Almeida, Telmo Matos and Jose Reis (CVRVV) @ 2009, for the research published in [CCA+09], to which we refer.

Arrhythmia Cardiac Arrhythmia Database, Sources:
Original owners od Database:
– 1. H. Altay Guvenir, PhD., Bilkent University, Department of Computer Engineering and Information Science, 06533 Ankara, Turkey Phone: +90 (312) 266 4133 Email: guvenir@cs.bilkent.edu.tr
– 2. Burak Acar, M.S., Bilkent University, EE Eng. Dept. 06533 Ankara, Turkey Email: buraka@ee.bilkent.edu.tr
– 3. Haldun Muderrisoglu, M.D., Ph.D., Baskent University, School of Medicine Ankara, Turkey
Donor: H. Altay Guvenir Bilkent University, Department of Computer Engineering and Information Science, 06533 Ankara, Turkey Phone: +90 (312) 266 4133 Email: guvenir@cs.bilkent.edu.tr
Date: January, 1998
For past data-set usage, we refer to [GK14].

Dermatology Dermatology Database, Source Information:
Original owners:
– 1. Nilsel Ilter, M.D., Ph.D., Gazi University, School of Medicine 06510 Ankara, Turkey Phone: +90 (312) 214 1080
– 2. H. Altay Guvenir, PhD., Bilkent University, Department of Computer Engineering and Information Science, 06533 Ankara, Turkey Phone: +90 (312) 266 4133 Email: guvenir@cs.bilkent.edu.tr
Donor: H. Altay Guvenir, Bilkent University, Department of Computer Engineering and Information Science, 06533 Ankara, Turkey Phone: +90 (312) 266 4133 Email: guvenir@cs.bilkent.edu.tr
Date: January, 1998.

Boston Housing Data Sources:
Origin: This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.
Creator: Harrison, D. and Rubinfeld, D.L. 'Hedonic prices and the demand for clean air', J. Environ. Economics Management,vol.5, 81-102, 1978.
Date: July 7, 1993.

We refer to [HJR78] in Literature.

Bibliography

[AKE22] Syed Adnan Akhtar, Arman Sharifi Kolarijani, and Peyman Mohajerin Esfahani. Learning for control: An inverse optimization approach. *IEEE Control Systems Letters*, 6:187–192, 2022.

[BCG+04] RK Bock, A Chilingarian, M Gaug, F Hakl, Th Hengstebeck, M Jiřina, J Klaschka, E Kotrč, P Savickỳ, S Towers, et al. Methods for multidimensional event classification: a case study using images from a cherenkov gamma-ray telescope. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 516(2-3):511–528, 2004.

[Bem06] Alberto Bemporad. Model predictive control design: New trends and tools. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 6678–6683. IEEE, 2006.

[BMPS20] Andreas Bärmann, Alexander Martin, Sebastian Pokutta, and Oskar Schneider. An online-learning approach to inverse optimization, 2020.

[BPS17] Andreas Bärmann, Sebastian Pokutta, and Oskar Schneider. Emulating the expert: Inverse optimization through online learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 400–410. PMLR, 06–11 Aug 2017.

[CCA+09] Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. Modeling wine preferences by data mining from physicochemical properties. *Decision support systems*, 47(4):547–553, 2009.

[DE83] Persi Diaconis and Bradley Efron. Computer-intensive methods in statistics. *Scientific American*, 248(5):116–131, 1983.

[ESAHK17] Peyman Mohajerin Esfahani, Soroosh Shafieezadeh-Abadeh, Grani A. Hanasusanto, and Daniel Kuhn. Data-driven inverse optimization with imperfect information. *Mathematical Programming*, 167(1):191–234, December 2017.

[GK14] Giulia Guidi and Manas Karandikar. Classification of arrhythmia using ecg data. *Lecture notes*, 2014.

[HJR78] David Harrison Jr and Daniel L Rubinfeld. Hedonic housing prices and the demand for clean air. *Journal of environmental economics and management*, 5(1):81–102, 1978.

[KBP13] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[Lee11] Jay H Lee. Model predictive control: Review of the three decades of development. *International Journal of Control, Automation and Systems*, 9(3):415–424, 2011.

[LKYC16] Hoang Le, Andrew Kang, Yisong Yue, and Peter Carr. Smooth imitation learning for online sequence prediction. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 680–688, New York, New York, USA, 20–22 Jun 2016. PMLR.

[Lof04] J. Lofberg. Yalmip : a toolbox for modeling and optimization in matlab. In *2004 IEEE International Conference on Robotics and Automation (IEEE Cat. No.04CH37508)*, pages 284–289, 2004.

[L.V13] L.Vandenberghe. Piecewise-linear optimization, 2013. Lecture notes at UCLA Samueli University.

[Mac02] M.J. Maciejowski. *Predictive Control with Constraints*. Pearson Education, 2002.

[Mat20] Stephan Matzka. Explainable artificial intelligence for predictive maintenance applications. In *2020 Third International Conference on Artificial Intelligence for Industries (AI4I)*, pages 69–74. IEEE, 2020.

[Nem04] Arkadi Nemirovski. Prox-method with rate of convergence o(1/t) for variational inequalities with lipschitz continuous monotone operators and smooth convex-concave saddle point problems. *SIAM Journal on Optimization*, 15(1):229–251, January 2004.

[Sch87] Jeffrey Curtis Schlimmer. *Concept acquisition through representational adjustment*. PhD thesis, University of California, Irvine, 1987.

[SSK15] Kamaljot Singh, Ranjeet Kaur Sandhu, and Dinesh Kumar. Comment volume prediction using neural networks and decision trees. In *IEEE UKSim-AMSS 17th International Conference on Computer Modelling and Simulation, UKSim2015 (UKSim2015)*, 2015.

[SWM93] W Nick Street, William H Wolberg, and Olvi L Mangasarian. Nuclear feature extraction for breast tumor diagnosis. In *Biomedical image processing and biomedical visualization*, volume 1905, pages 861–870. International Society for Optics and Photonics, 1993.

[WM90] William H Wolberg and Olvi L Mangasarian. Multisurface method of pattern separation for medical diagnosis applied to breast cytology. *Proceedings of the national academy of sciences*, 87(23):9193–9196, 1990.