Decision-making algorithms for autonomous robots

Ludovic Hofer

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Ludovic Hofer**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Decision-making algorithms for autonomous robots

Date de soutenance : 27 novembre 2017

Devant la commission d'examen composée de :

Olivier Ly	Maître de conférences, Université de Bordeaux	Directeur
Hugo Gimbert	Chargé de recherche, LaBRI	Co-directeur
Luca Iocchi	Professore associato, Sapienza Università di Roma	Rapporteur
Frédérick Garcia	Directeur de recherche, INRA	Rapporteur
Blaise Genest	Chargé de recherche, IRISA	Examinateur
Laurent Simon	Professeur, Bordeaux INP	Examinateur
Serge Chaumette	Professeur, Université de Bordeaux	Président du jury
Abdel-Illah Mouaddib	Professeur, Université de Caen	Examinateur

– 2017 –

Titre Algorithmes de prise de décision stratégique pour robots autonomes

Résumé Afin d'être autonomes, les robots doivent êtres capables de prendre des décisions en fonction des informations qu'ils perçoivent de leur environnement. Cette thèse modélise les problèmes de prise de décision robotique comme des processus de décision markoviens avec un espace d'état et un espace d'action tous deux continus. Ce choix de modélisation permet de représenter les incertitudes sur le résultat des actions appliquées par le robot.

Les nouveaux algorithmes d'apprentissage présentés dans cette thèse se focalisent sur l'obtention de stratégies applicables dans un domaine embarqué. Ils sont appliqués à deux problèmes concrets issus de la RoboCup, une compétition robotique internationale annuelle. Dans ces problèmes, des robots humanoïdes doivent décider de la puissance et de la direction de tirs afin de maximiser les chances de marquer et contrôler la commande d'une primitive motrice pour préparer un tir.

Mots-clés Processus de décision markovien, robotique autonome, apprentissage

Abstract The autonomy of robots heavily relies on their ability to make decisions based on the information provided by their sensors. In this dissertation, decision-making in robotics is modeled as continuous state and action markov decision process. This choice allows modeling of uncertainty on the results of the actions chosen by the robots.

The new learning algorithms proposed in this thesis focus on producing policies which can be used online at a low computational cost. They are applied to real-world problems in the RoboCup context, an international robotic competition held annually. In those problems, humanoid robots have to choose either the direction and power of kicks in order to maximize the probability of scoring a goal or the parameters of a walk engine to move towards a kickable position.

Keywords Markov decision process, Autonomous robotics, Machine learning

Laboratoire d'accueil LaBRI, Bâtiment A30, 351, Cours de la Libération 33405 Talence CEDEX, France

Résumé de la thèse en français

Alors qu'en mai 1997, Deep Blue était le premier ordinateur à vaincre le champion du monde en titre aux échecs, signant ainsi une victoire majeure de l'intelligence artificielle sur celles des humains, la RoboCup naissait. Cette compétition internationale de robotique affichait l'objectif ambitieux de produire une équipe de robots humanoïdes capables de remporter un match de football contre l'équipe humaine championne du monde. Environnement partiellement observable, espaces continus et coopération, les difficultés théoriques présentes dans cette compétition ont forcés les participants à passer d'une résolution de problèmes symboliques à une résolution de problèmes concrets où les résultats des actions ne correspondent parfois pas aux attentes. Cette compétition a grandi au cours des 20 dernières années et elle rassemble à présent plus de 3000 participants à chaque année.

Au cours de ma thèse, j'ai participé à trois reprises à la RoboCup au sein de l'équipe Rhoban dans la catégorie "robots humanoïdes de petite taille". Cette thèse étudie plusieurs problèmes présents dans le football robotique humanoïde et présente des algorithmes permettant d'y apporter des solutions.

Afin d'être autonomes, les robots doivent être capables de percevoir leur environnement en analysant les informations reçues par leurs capteurs. Dans le cas des robots footballeurs de la RoboCup, ils ne perçoivent leur environnement qu'à travers une caméra, une centrale inertielle et des capteurs angulaires leur permettant de mesurer la position angulaire de leurs joints.

Les robots sont aussi limités par leurs capacités d'interactions avec leur environnement. La marche robotique bipède est elle-même un sujet de recherche à part entière. Réaliser un tir puissant est aussi une tâche complexe puisqu'elle nécessite de conserver l'équilibre tout en effectuant un mouvement hautement dynamique.

Alors que les robots industriels effectuent des tâches répétitives, les robots autonomes ont des objectifs et cherchent à les accomplir. Pour ce faire, ils décident d'appliquer les actions appropriées à la représentation qu'ils se font de leur environnement. Cette prise de décision est l'axe de recherche de cette thèse.

Puisqu'ils se déplacent et agissent dans le monde réel, les robots ne peuvent pas se permettre de négliger le fait que le résultat de leurs action n'est pas déterministe en fonction de leur perception. Ils doivent donc être capable de

prendre des décisions dans des environnements stochastiques, établissant des stratégies leur permettant d'atteindre leurs objectifs malgré la difficulté de prédire le résultat de leurs actions.

Pour refléter à la fois les aspects stochastique et continus des problèmes robotiques, cette thèse modélise les problèmes comme des processus de décision Markoviens à espaces d'état et d'action continus, abrégés PDM-EAC.

La prise de décision en robotique est sujette à des contraintes spécifiques. En plus de devoir prendre en compte les aspects stochastiques et continus des problèmes, elle doit s'exécuter en temps réel sur des processeurs parfois peu puissants. Effectivement, les contraintes en termes de poids, de volume et de consommation énergétique qui s'appliquent en robotique sont moins prononcées pour d'autres champs d'application.

Cette thèse propose trois algorithmes permettant d'optimiser des stratégies pour des PDM-EAC. Elle étudie aussi en profondeur deux sous-problèmes auxquels notre équipe a fait face lors des différentes édition de la RoboCup: *l'approche de la balle* et *le choix du tir*.

Dans le problème de *l'approche de la balle*, un robot doit choisir quels ordres envoyer à sa primitive motrice pour atteindre une position convenable pour effectuer son prochain tir au plus vite. En plus de minimiser le temps nécessaire à atteindre la balle, le robot doit s'assurer d'éviter d'entrer en collision avec la balle. Cette tâche est particulièrement difficile car il est difficile de prévoir les déplacements exacts en fonction des ordres envoyés. Le bruit est particulièrement important car les robots se déplacent sur de la pelouse artificielle et glissent légèrement sur le sol, rendant ainsi le positionnement précis difficile. Dans le cadre de la compétition, cette tâche est critique, car les robots mettent souvent plus de temps à effectuer les 50 derniers centimètres nécessaires à leur placement qu'à parcourir les 3 mètres précédant cette phase délicate.

Lors des matchs de football humanoïdes, les robots doivent coopérer pour choisir quel robot va tirer, dans quel direction et avec quel type de tir. Nous appelons ce problème *le choix du tir*. S'il semble évident qu'il vaut mieux tirer en direction des cages adverses, il n'est pas simple de choisir s'il vaut mieux passer la balle à un partenaire, tirer directement dans le but où simplement recentrer la balle.

Le problème du tir revêt un aspect particulier, parmi les choix qui s'offrent au robot, il y a plusieurs actions distinctes, chacun d'entre elle ayant des paramètres continus. Pour modéliser cet aspect du problème, cette thèse introduit la notion d'espace d'action hétérogène et présente un nouvel algorithme permettant de trouver des stratégies efficaces pour des problèmes avec espace d'action hétérogène.

Les espaces d'états et d'actions étant continus, ils sont infinis. Afin de pouvoir représenter certaines de leurs propriétés, cette thèse se base sur des approximateurs de fonction et plus particulièrement sur les arbres de régressions et les forêts de régressions.

Les arbres de régressions représentent une fonction à l'aide d'un arbre où chaque noeud sépare l'espace d'entrée à l'aide d'un prédicat. Dans cette thèse, nous nous restreignons aux prédicats se présentant sous la forme d'une inégalité sur une seule dimension. Les feuilles d'un arbre de régression contiennent des modèles permettant d'attribuer une prédiction à chaque élément d'entrée. Seuls les modèles linéaires et constants sont considérés ici.

Les forêts de régressions sont des ensembles d'arbres de régression. Elles permettent d'agréger les prédictions faites par différents arbres pour améliorer sa qualité.

Le premier algorithme proposé par cette thèse est *fitted policy forests*, abrégé FPF. Cet algorithme permet d'apprendre une stratégie représentée par une forêt de régression à partir d'échantillons récolté lors d'interactions avec le système. Les résultats obtenus par FPF sur des problèmes classiques d'apprentissage par renforcement sont satisfaisants par rapport à l'état de l'art et l'utilisation des stratégies en temps réel est éprouvée sur ces mêmes problèmes.

Malgré les performances satisfaisantes de FPF sur les problèmes classiques, il semble difficile de l'appliquer à des cas robotiques car il demande un nombre d'échantillons trop important pour atteindre des performances satisfaisantes. Cette constatation nous a mené à changer de paradigme et à ne pas chercher à apprendre une stratégie directement sur le robots, mais plutôt à modéliser le problème comme une boîte noire avec des paramètres. Un nombre restreint d'échantillons sont ensuite acquis dans le monde réel pour calibre les paramètres de la boite noire et ainsi permettre d'avoir un modèle réaliste qui puisse être utilisé pour entraîner des stratégies sans nécessiter d'interaction avec le robot.

L'accès à un modèle boite noire du problème nous a permis de développer un second algorithme nommé *random forests policy iteration*, abrégé RFPI. Cet algorithme profite du modèle en boite noire pour optimiser des stratégies représentées par des forêts de régression. Il est appliqué au problème de l'approche de la balle avec succès produisant des stratégies plus performantes que celles qui nous ont permis de gagner lors de la RoboCup 2016. Ces résultats étant obtenus par validation expérimentale, à la fois en simulation et dans le monde réel.

Bien qu'il ait produit des stratégies efficaces sur le problème de l'approche de la balle, RFPI ne permet pas de résoudre le problème du choix du tir car notre implémentation de cet algorithme ne permet pas de traiter les espaces d'actions hétérogènes.

Afin de pouvoir apprendre des stratégies efficaces pour le problème du choix du tir, nous proposons un troisième algorithme nommé *Policy Mutation Learner* et abrégé PML. Il représente à tout moment la stratégie comme un arbre de régression et effectue des mutations des feuilles ayant un impact local sur la stratégie, mais un impact global sur les performances. Grâce à ce

mécanisme de mutation, PML s'assure que les modifications de la stratégies sont refusées si elles nuisent aux performances. Ce point précis fait de PML un algorithme particulièrement adapté à l'amélioration de stratégie experte déjà existantes.

Les résultats expérimentaux de PML au problème du choix du tir en simulation ont montré que PML était capable aussi bien d'améliorer significativement les performances de stratégies existantes que d'en créer sans recevoir d'informations initiales.

En comparant le temps moyen nécessaire pour marquer un but entre le problème du choix du tir à deux joueurs et son équivalent à un joueur. Nous avons montré que jouer en équipe à un intérêt malgré le bruit et l'absence de détection des adversaires, un fait qui n'était pas évident au vu de l'importance du bruit sur le tir et donc de la difficulté d'estimer la position de la balle après le tir.

Une analyse plus poussée des gains apportés par le jeu d'équipe a mis en exergue les imperfections de PML. Bien que cet algorithme améliore de manière significative les stratégies, il ne parvient pas toujours à sélectionner le bon joueur pour tirer quand il y a plusieurs candidats. L'identification de cette faiblesse permet d'envisager plusieurs modifications susceptibles d'améliorer considérablement les résultats obtenus dans le futur.

Acknowledgements

Firstly, I would like to express my gratitude to my two Ph.D advisors Olivier Ly and Hugo Gimbert for their support. They accepted me as a member of their RoboCup team while I was studying in bachelor. Since then, I shared many other robotics adventure with them and they always trusted in my capability of being autonomous, from internships until the end of my dissertation.

Besides my advisors, I would like to thank all the members of the jury for their insightful comments and questions which have widen my research interests.

My sincere thanks also goes to Pascal Desbarats who provided insightful advices and a strong support during my thesis.

I would also like to thanks specifically present and past members of the Rhoban team. Loic gondry for his amazing work on the mechanics of the robots. Grégoire Passault for the infinite amount of energy he invested in the team and his brutal pragmatism that force us to perform self criticism, he opened many interesting discussions and always faced problems looking forward with an unstoppable enthusiasm. Quentin Rouxel for his friendship, his contagious enthusiasm for the newest c++ features and his zealous support for establishing the required norms. Finally, I would like to thank Steve N'Guyen who brought so much to me during this thesis: from harsh realism to an exceptional academical guidance. He has simply been there, helping with an unmatched modesty.

A special thank to my family who helped me to hold on during the hardest time of my PhD. Last but not least, words cannot express how grateful I am to Marine Guerry, she knows when and how to comfort me, but she is also able to step-back when required. She helped me more than she will ever know.

Contents

Introduction

This chapter describes the main components involved in the decision-making process of autonomous robots while targeting a general audience. It discuss the need of approaching real robotic problems to identify the specificities of decision-making in robotics, present various aspects of the RoboCup competition and then briefly outlines the contributions of this thesis.

RoboCup

In May 1997, IBM Deep Blue defeated the world champion in chess. While this is a major milestone for artificial intelligence, chess is an adversarial discrete game with perfect information. Therefore, the program who won did not have to deal with uncertainties, continuous spaces or team-play. This year, the first official RoboCup was held. This competition had three soccer leagues, one in simulation and two with wheeled robots of different sizes. The long-term goal of the RoboCup initiative is to have a team of fully autonomous humanoid robot winning a soccer game against the winner of the most recent soccer World Cup. This challenge involves complex perception and actuation in robotic, but also decision-making and cooperation between robots.

Twenty years after the creation of the RoboCup, the competition has grown and gathers around 3000 researchers every year with more than 10 different leagues, including other challenges than soccer such as performing rescue tasks or helping in daily-life situations.

RoboCup helps to identify key challenges of autonomous robotics, it allows to evaluate the performance of robots on complex situations and promote collaboration between researchers from different backgrounds. It also provide a large field to test the applicability of theories and models to real-world problems.

During this thesis, I participated to the RoboCup in the KidSize Humanoid League three times with the Rhoban Football Club. An example of game situation is shown in Fig. 1 We ranked 3rd in 2015 and 1st in 2016 and 2017. In the KidSize Humanoid League, two teams of four humanoid robots play soccer against each other. There are many restrictions on the design of the robots. We cite here a few important limitations:

Figure 1 – A KidSize Humanoid League soccer game at RoboCup 2017

- The height of the robot is between 40 and 90 centimeters.
- Active sensors based on light and electromagnetic waves are forbidden.
- The maximum number of camera is 2.
- The field of view of the robot is limited at 180 degrees.

We present here the three most important challenges we identified in KidSize league. First, the locomotion on artificial grass: the robots must be able to walk on a 3 centimeters height grass which can be assimilated to soft ground, moreover, it has to be capable of falling and standing up without damaging himself. Second, real-time perception: the robots have to localize themselves and detect the ball using a camera mounted on the head of the robot, this is particularly difficult because the computational power is limited and the head is shaking while the robot is walking. Finally, decision making in stochastic environments: due to the perception noise and the artificial turf, it is difficult to predict the result of the robot actions. Therefore, the robot must learn how to act given uncertainty.

The humanoid robot we used for our experiments and the RoboCup is named Sigmaban, see Figure 2. It has 20 degrees of freedom controlled in position at a frequency of approximately 100 Hz. Acquisition of information is mainly performed through online computer vision, using 640x480 images, additional information are provided by angular sensor in each joint, an inertial measurement unit in the hip and pressure sensors in the feet. Due to the low amount of computational power available on board and the fact that the head

of the robot is shaking during image acquisition, the quality of the localization of the robot inside the field and its estimation of the ball position presents an important noise.

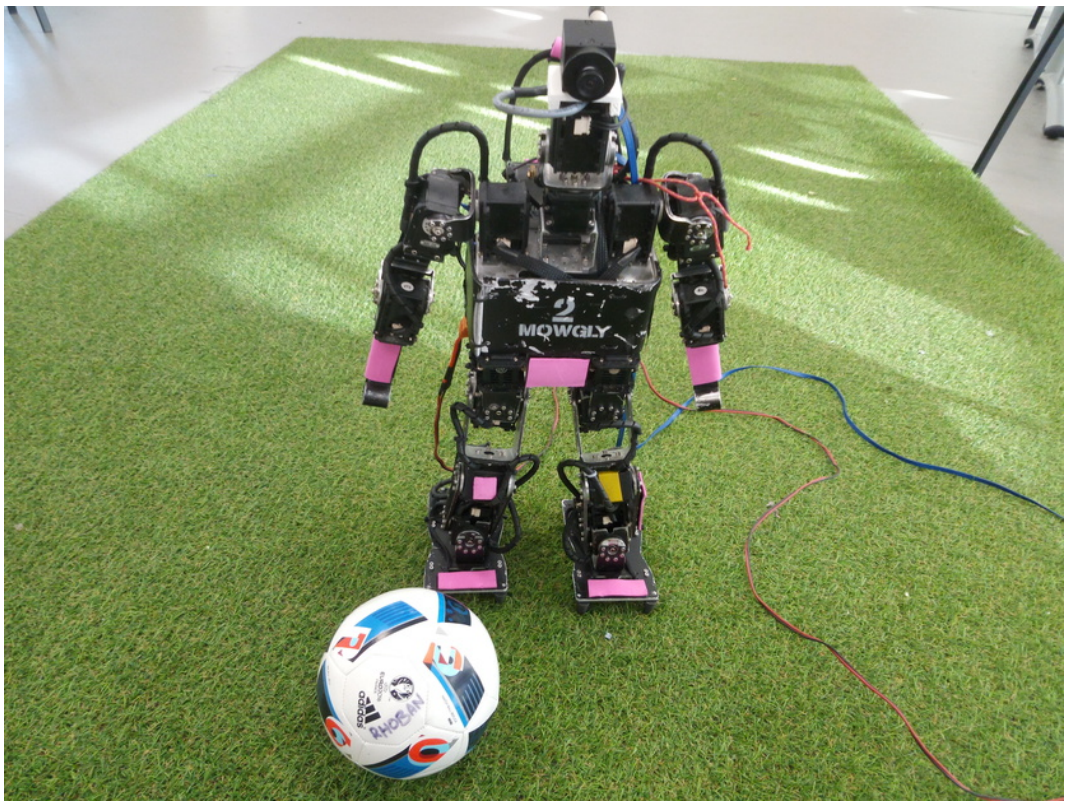


Figure 2 – The Sigmaban robot

Decision making for autonomous robots

Autonomous robots have to adapt their decisions to their environments. They use their sensors and an internal memory to analyze and update their representation of both, the state of the world and their inner state. In order to estimate the speed of an object, robots will usually accumulate data from a distance sensor. By observing the evolution of the distance, robots will be able to estimate the speed of the object. Perception of the environment by robots is a complex problem which will not be treated in this thesis. However, it is important to keep in mind that the decisions made by robots are based on partial and noisy information. Readers interested on the perception problem can refer to [McCallum 1996], a very insightful thesis on the problem of selective perception and hidden states.

Decision-making is the process of choosing the action a robot should perform given its inner representation of the world state. This process can simply

be applying an existing policy, i.e a function mapping every possible world state to an action, but it can also involve simulations in order to predict the results of possible actions in the current situation. In both cases, the aim of the decision process is to maximize the reward received by the robot.

There are three important aspect to consider for decision-making in robotics. First, autonomous robots have to embed their computational power, due to constraints on weight, volume and energy consumption, therefore decision making should be computationally efficient. Second, decision-making has to be quick in order to avoid delay, e.g. an autonomous car has to react quickly to avoid collision. Finally, it is important to keep in mind that models used in robotics might be approximation of the real world. There is generally a significant gap between estimations provided by the physical model (theory) and the consequences of its actions in the real-world (practice). This discrepancy is particularly high on for low-cost robots of the kind we are interested in. Therefore, acting optimally according to a model and acting optimally in the real world are two different things.

This thesis aims at providing solutions on two aspects of decision-making in robotics. How to model a decision-making problem and how to train lightweight policies which can be used online.

From theory to practice

Developing complex and efficient behaviors on robots generally requires the use of simulators in order to assess the performances without risking any damage on the robot. Simulation allows:

1. Optimizing control functions by experimenting various policies, eventually in parallel.

2. Monitoring values which cannot be measured accurately on robots.

Simulations are based on physical equations and specifications of the robots. There might be a significant gap between reality and the simulator. Some of the physical aspects such as chocs are difficult to model accurately, especially for low-cost robots built with low-cost hardware. The real robot may slightly differs from the model due to mechanical imperfections and incomplete modeling (e.g. the weight of the electrical wires is rarely included in the model of robots and every servomotor has a different transfer function). While it is possible to reduce the gap between model and reality it often has a significant cost, because it requires more accurate manufacturing.

Perception of the environment by the robots is subject to noise. All the digital sensors present a quantification noise due to the resolution of the sensors. Sensors also provide information at a given frequency, therefore, it is not possible to have access to the robot status at any time. As a consequence,

sensors do not provide exact measurements of a physical parameter, but rather a probability distribution over its possible values.

Robots cannot capture entirely and accurately neither their internal state or the environment status. They might lack information about the friction coefficient of the floor, the exact position of an encoder or have a rough accurate of their position in their environment. Therefore, applying the same action in the same measured state can lead to different results depending on non-measured variables.

Since decision making in partially observable environments is particularly difficult, we choose to consider robotic problems as fully observable but with stochastic transitions. While this type of abstraction does not reflect reality, it allows the use of efficient algorithms and still leads to satisfying results.

While increasing the accuracy of the robot hardware makes decision-making easier and leads to improvements on the ability to perform most of the tasks, it also increases the cost of the robot. In order to reach satisfying performances at a low-cost, it is necessary to take into account the noise in the decision-making process. Moreover, modeling the noise of the robot and using learning to solve the problem also provides guidelines for hardware changes by automatically detecting which aspects of the noise have the highest impact on performance.

Additionally to all the difficulties mentioned previously, online decision making for autonomous robots needs to consider specific constraints regarding computational power. Autonomous robots need to embed their computational power with three major limitations: volume, weight and power consumption. Moreover, since robots act in real world, their decision-making has to be real-time. Therefore, decision making in autonomous robotics often require to focus on providing acceptable strategies at a low computational cost.

Applications of learning to robotics

Reinforcement learning has been used successfully in robotics for the last two decades. While section present key concepts of reinforcement learning and their relationship with robotics, this section presents a few successful applications with some context.

Autonomous radio-controlled helicopter flight was achieved in [Ng *and al.* 2004]. The authors started by fitting a model using locally weighted linear regression [Atkeson *and al.* 1997] and then used their model to train a policy based on the reinforcement learning algorithm PEGASUS [Ng and Jordan 2000]. While the state space has 12 dimensions and the action space 4, they still managed to produce a policy which strongly outperforms specially trained human pilots with respect to stability. Moreover, they were also able to use their model to perform maneuvers from challenging RC competitions.

Hitting a baseball with an anthropomorphic arm was performed in [Peters and Schaal 2006]. The main difficulty of this task relies on its strong

dependency to the model dynamics which are particularly sensitive to small modeling errors. In order to achieve this performance, the authors started by teaching a simple stroke with supervised learning. However, the robot was not able to reproduce the presented stroke. Therefore, they used a policy gradient algorithm to bridge the gap between the demonstration and an appropriated move. After 200 to 300 rollouts, the robotic arm was able to hit the ball properly.

Several researchers have managed to perform the ball-in-a-cup task using robotic arms [Kober *and al.* 2009; Nemec *and al.* 2010]. In [Nemec *and al.* 2010], the authors experimented two different methods to achieve this task. One based on dynamical movement primitives [Ijspeert *and al.* 2013] which uses human demonstration, and the other one based on reinforcement learning without prior knowledge of the system. The second one start by using around 300 simulations to perform an offline training and then continues learning on the real robot, ending up with an appropriate policy for the real world after 40 to 90 additional rollouts.

Learning how to balance an unstable pole on a free angular joint by driving a cart on a linear track after only a few rollouts was achieved in [Deisenroth and Rasmussen 2011]. The model of the system was learned on the real robot, using Gaussian processes [Rasmussen 2006]. Learning an accurate model from a few samples was a key to the success of this method.

For further reading, a thorough review of the applications of reinforcement learning in robotics is presented in [Kober and Peters 2012].

All the applications mentioned here are based on learning policies for dynamic and continuous problems with a high-quality hardware. This thesis aims at providing learning algorithms for problems whose underlying dynamic involves impacts. Moreover, we focus on applying learning for high-level tasks with low-cost hardware, thus increasing the amount of noise faced.

Problems for humanoid soccer-playing robots

Algorithms presented in this thesis have been experimented on humanoid robots in the framework of the RoboCup, see . In this thesis, we focus on solving two problems, the *ball approach* and the *kicker robot* problems.

The global problem of playing soccer is to control the position of each joint of the four robots at 100 Hz according to the inputs of all the sensors, while targeting to score goals. This would lead to an action space with 80 continuous dimensions and even more state dimensions. Since learning how to walk is already a complex task for humanoid robots, solving this problem is intractable. Therefore we rely on expert algorithms for locomotion and perception. The decision-making problem we consider is: which orders should the robots send to the walk engine and how should they kick to score a goal as quickly as possible against a static goalie. Since this problem is still quite

complex, we used a decomposition of the problem in two different tasks. First, what orders should a robot send to the walk engine to prepare a kick in a given direction. Second, which kick should a robot perform to score as quickly as possible.

In the *ball approach* problem, a robot tries to reach a suitable position to perform a kick. It uses an expert holonomic walk engine. The aim of the robot is to reach the suitable position as quickly as possible, while avoiding collisions with the ball. Since the robot is walking on artificial turf and the control of the motors is not accurate, every step of the robot include some noise on its next position and orientation. We establish an appropriate model for this problem in section 1.2.6. In chapter 3, we train policies beating expert strategies by using reinforcement learning algorithms.

In the *kicker robot* problem, see 1.3, Robots are playing on a field with a single static goalie as opponent. This problem has many parameters among which we find the number of players. A central agent has to choose which robot will kick and how it will perform the kick by choosing some parameters, e.g. direction of the kick. The aim is to find the quickest policies to score goals reliably. The robot might fail to score a goal if it shoots outside of the field or if the ball collides with the goalie. This problem also allows to assess the gain of playing as a team versus playing individually. Chapter 4 presents experimental results for this problem along with considerations about the parametrization of the problem. It shows how it might be more efficient to optimize policies on approximated models and presents the advantages of playing in cooperation with other robots of the team.

Reinforcement learning

In reinforcement learning problems, the goal is to learn a policy which maximizes the expected reward (or minimizes the expected cost). The learning algorithm has only access to a reward function and a method to gather samples, either by interacting with the system or by sampling a stochastic function. There is no human supervision during the learning process.

The main challenges of learning

Since the dawn of reinforcement learning, it has been applied on a wide variety of problems. A question that arise is: "What makes a problem difficult to solve for reinforcement learning?". In this part, we will discuss several recurring issues in this field and their impact on the approaches used to solve the problems.

Exploitation vs exploration

While interacting with a system, agents gather data on the problem they are trying to solve. At anytime, they can exploit this knowledge

to build a policy which allows them to maximize the expected reward considering their current belief. However, if an agent always tries to act optimally in its space, it can end up repeating similar actions and therefore its knowledge of the problem stays limited. On the other side, if an agent explore continuously, it will never exploit the data acquired. We refer to this problem as the "exploitation versus exploration trade-off". A well known problem where the agent has to handle the trade-off between exploitation and exploration is the multi-armed bandit, see [Gittins *and al.* 1979].

Smart exploration

Exploring using random actions has proved to be sufficient for some problems but very ineffective on other situations, particularly when the state space is large or when some states can be reached only after applying a specific sequence of actions. Some theoretical problems such as the *combinatory lock problem* have been presented to highlight the necessity of using smart exploration. In the combinatory lock problem, there is n states $(1, \ldots, n)$ and the agent start in state 1. In every state he can either choose action a which leads to state $k+1$ from any state k or action b which always lead to state 1. In order to reach state n, the agent has to apply n consecutive actions a, if it uses random exploration, the probability of reaching state n after each sequence of n actions is $\frac{1}{2^n}$. Therefore, even with a simple system with only 50 states and two deterministic actions, random exploration might already require around 10^{15} actions to reach the target state. The discrete case of the combinatory lock problem appears in [Koenig and Simmons 1996] and the continuous case appears in [Li *and al.* 2009]. In this type of problem, exploration should provide an incentive for applying actions which are likely to lead to states which have been less frequently visited. If we use this type of incentive on the combinatory lock problem, action a will quickly become preferable to b in every visited state, because action b leads to states which have been visited frequently.

Local and global maxima

Consider a function $f : X \mapsto \mathbb{R}$ with $X \in \mathbb{R}^n$, $x \in X$ is considered as a local maximum of f if and only if there is an $\epsilon \in \mathbb{R}^+$ such as $\forall x' \in X, \|x' - x\| < \epsilon \Rightarrow f(x') \leq f(x)$. These notions can be extended to policies by considering that a policy is locally optimal if in every state, the action to be applied is a local maxima with respect to the current policy. While optimizing functions or policies one of the major risks is to end up stuck in a local maxima. This issue generally arises when using gradient-based methods or when the mean value in the neighborhood of the global maxima is lower than the value

around a local maximum. A simple theoretical example might be the following, the agent ride a bicycle and he needs to cross a river. There are two bridges, one very narrow but nearby and another very wide but further. While exploring different trajectories, the agent will likely fail several times at crossing the narrow bridge and succeed quickly at crossing the wide bridge. Therefore, even if the optimal policy passes through the narrow bridge, learning algorithms might get stuck on a policy passing through the wide bridge.

Curse of dimensionality

The curse of dimensionality often arises while learning policies, optimizing functions or building policies. The problem is that when solving a problem with n dimensions, the number of required interactions with the system grows exponentially with n. This problem might concern the number of samples required for discretization or regression, it might also concern the number of calls required to find the global maxima while optimizing a function with a multi-dimensional output. Therefore, some schemes which are effective in low-dimensional problems might be computationally intractable when scaling up to problems with a higher number of dimensions

Overfitting vs underfitting

While building a model from observations, it is usual to have different types of models such as linear models, quadratic models or Gaussian mixtures. Those models have generally a set of parameters and the learning process tries to reduce the error between the model and the observations. Obviously, more expressive models will allow to reduce the error between the model and the observations. But when there is noise involved, either on the observation or even on the process observed, even if the model matches perfectly the observations used to build it, it is possible that the difference between new observations and their prediction according to the model might be large. The problem of having a much lower error on the training observations than during validation is called overfitting. It is very likely to arise when the number of samples gathered is low with respect to the number of parameters of the model or when the model is more complex than reality.

Online and offline learning

While they share a common goal, online and offline learning have different constraints and advantages and cannot be compared directly. In this part we the notion of blackbox function is introduced and then four different types of

reinforcement learning are presented: online, offline, batch mode and semi-online.

While sampling a CSA-MDP on a robot, it is not possible to choose the initial state. Moreover, acquiring real-world samples is very time consuming. Therefore, it is often convenient to build a blackbox model of the transition function. In this thesis we consider that blackbox models can involve stochastic elements, therefore, they can produce various outputs for the same input.

We denote $\mathcal{B} : I \mapsto \Delta(O)$ a blackbox $\mathcal{B}$ with input space I and output space O, where $\Delta(O)$ denotes the set of probabilities densities on O. We write the process of sampling the blackbox $o = \text{sample}(\mathcal{B}(i))$, with $i \in I$ and $o \in O$. For the specific case of MDPs, we have $I = S \times A$ and $O = S \times \mathbb{R} \times \{\dashv, \vdash\}$.

Online learning focuses on experimenting in real situation. It requires the possibility of adding new information at a low computational cost, otherwise the learning system would not be able to treat the observations quickly enough. It presents the advantage of adapting directly to the robot and therefore it can adapt to specificities of the robot which would not be taken into account in expert models of the system.

Offline learning is based on an existing transition model and uses symbolic representation or blackbox simulation to compute efficient policies. It benefits from more computational resources than online learning since it has no real-time constraint. However, requesting the best action from computed policies should be possible in real-time.

Batch mode learning is entirely based on samples collected through interaction with the real system. It happens in two phases, first all the samples are gathered, then they are used to train a policy. It is possible to use batch mode algorithms online, by updating the policy regularly. However, batch mode learning focus on exploitation, therefore it has no mechanism to ensure exploration.

Semi-online learning is based on the notion of rollout. During a rollout, the agent will act according to the current policy. Once the rollout is finished, the agent uses the samples gathered during the rollout to improve current policy. Thus, the update does not require to be real-time and it can even be performed on external hardware.

Episodic learning

In some problems, the initial state of the system is known and the target is always the same. A specific aspect of those problem is the fact that it is not required to explore the whole state space in order to provide an efficient solution. Therefore, the most efficient approaches for episodic learning are generally based on exploring only a subspace of the entire state space. Classical episodic learning problems are *Car on the hill*, see 1.1.2 and *Inverted pendulum stabilization*, see 1.1.3.

Markov decision processes

Markov decision processes, MDP for short, is a framework that allows modeling of problems including non-deterministic environments. Originally, MDP were designed to express problems with a finite number of states and actions. However, most real-life problem are continuous, therefore other models have been proposed which use continuous states and actions, CSA-MDP for short. Unless explicitly stated otherwise, problems mentioned in this thesis will be CSA-MDP.

Definitions

An hyperrectangle $\mathcal{H} \subset \mathbb{R}^n$ is a space defined by minimal and maximal boundaries along each dimension. $\mathcal{H}_{k,1}$ denote the minimum possible value for $\mathcal{H}$ along dimension k. $\mathcal{H}_{k,2}$ denote the maximum possible value for $\mathcal{H}$ along dimension k.

$$\mathcal{H} = \begin{bmatrix} \mathcal{H}_{1,1} & \mathcal{H}_{1,2} \\ \vdots & \vdots \\ \mathcal{H}_{n,1} & \mathcal{H}_{n,2} \end{bmatrix} = [\mathcal{H}_{1,1}\mathcal{H}_{1,2}] \times \cdots \times [\mathcal{H}_{n,1}\mathcal{H}_{n,2}] \tag{1}$$

We denote the volume of an hyperrectangle $|\mathcal{H}|$ and it is defined by equation 2.

$$|\mathcal{H}| = \prod_{k=1}^{n} \mathcal{H}_{k,2} - \mathcal{H}_{k,1} \tag{2}$$

The center of an hyperrectangle $\mathcal{H}$ is noted $\textsc{center}(\mathcal{H})$.

Before taking any decision, the agent can observe its current state $s \in S$, where S is the state space. All the state spaces considered in this thesis are hyperrectangles.

The agent has to choose both the action it will take, among a finite number of choices, and the parameters of the chosen action inside an hyperrectangle. We denote action spaces $A = (\mathcal{H}_{a_1}, \ldots, \mathcal{H}_{a_k})$, where $\mathcal{H}_{a_i}$ is the parameter space for the action with action identifier i. Parameters spaces for different actions can have a different number of dimensions.

At each step, the agent will receive a reward which depends on its state and the chosen action. The stochastic aspect of the reward received for applying action a in state s due to hidden parts of the transition process is discussed in section .

At each step, the agent receives a terminal status. The status $\vdash$ indicates that the agent can keep taking decisions, while the status $\dashv$ indicates that the agent cannot take any further decisions, either because it has reached a target or because it has encountered a critical failure.

We denote the transition function δ and we note $\delta(s', r, t|s, a)$ the density probability measure of reaching state $s' \in S$ with reward $r \in \mathbb{R}$ and terminal status $t \in \{\dashv, \vdash\}$

At the beginning of each task the agent is in a state sampled according to distribution $I = \Delta(S)$

We define a CSA-MDP by a five-tuple $\langle S, A, \delta, I, \gamma \rangle$, where:

- S is the state space.
- A is the action space.
- δ the transition function.
- I the initial state distribution
- $\gamma \in]0, 1]$ is the discount rate. It is generally used to decrease the impact of long-term reward.

Consider MDP $= \langle S, A, \delta, I, \gamma \rangle$ a CSA-MDP: the sampling of an initial state s according to I is denoted $s = \text{SAMPLEINITIALSTATE}(\text{MDP})$. The sampling of the result of a transition process from state s with action a according to δ is denoted $(s', r, \text{status}) = \text{SAMPLERESULT}(\text{MDP}, s, a)$, with $s' \in S$, $r \in \mathbb{R}$ and status $\in \{\dashv, \vdash\}$.

Consider MDP a CSA-MDP, we denote $\text{STATESPACE}(\text{MDP})$ its state space. We denote $\text{NBACTIONSPACES}(\text{MDP})$ the number of distinct action identifiers possible, i.e. the number of hyperrectangles in the action space. We denote the hyperrectangle defining the space of parameters allowed for action with action identifier $i \in \mathbb{N}$: $\text{ACTIONSPACE}(\text{MDP}, i)$. The discount rate of MDP is denoted $\text{DISCOUNT}(\text{MDP})$.

Policy: A deterministic policy $\pi : S \mapsto A$ defines which action $a \in A$ will be taken for any state $s \in S$. Thereafter, if it is not stated otherwise, by "policy", we implicitly refer to deterministic policy.

Value function: A value function $V^\pi : S \mapsto \mathbb{R}$ defines the value for each state $s \in S$ according to the policy π. This value function is the fixed point of Equation (3), denoted $V^\pi(s)$.

$$V^\pi(s) = \int_{s' \in S} \int_{r \in \mathbb{R}} \delta(s', r, \dashv | s, \pi(s))\, r + \delta(s', r, \vdash | s, \pi(s)) \left(r + \gamma V^\pi(s')\right) ds'\, dr \quad (3)$$

Q-Value: A Q-value $Q^\pi : S \times A \mapsto \mathbb{R}$ defines the expected reward when starting by applying action $a \in A$ in state $s \in S$ and then applying policy π. The value is the fixed point of Equation (4), denoted $Q^\pi(s,a)$.

$$Q^\pi(s,a) = \int_{s' \in S} \int_{r \in \mathbb{R}} \delta\left(s', r, \dashv | s, a\right) r + \delta\left(s', r, \vdash | s, a\right) \left(r + \gamma Q^\pi(s', \pi(s'))\right) ds'\, dr \tag{4}$$

Optimality: A policy π is considered as optimal if and only if it follows Equation (5). Optimal policies are denoted π^*.

$$\forall s \in S, \forall a \in A, V^{\pi^*}(s) \geq Q^{\pi^*}(s,a) \tag{5}$$

The value of a state is defined by $V(s) = \sup_\pi V^\pi(s)$, the best possible expected reward.

Solving MDPs

In this part, historical approaches used for solving finite-state MDPs and their applicability to continuous domain are discussed.

Value iteration

Value iteration algorithms are based on an iterative approach of the problem and try to estimate the value of each state using Equation (6).

$$V_{k+1}(s) = \max_{a \in A} \sum_{s' \in S} \int_{r \in \mathbb{R}} \delta\left(s', r, \dashv | s, a\right) r + \delta\left(s', r, \vdash | s, a\right) \left(r + \gamma V_k(s')\right) dr \tag{6}$$

The value function V_{k+1} is obtained by a single update step using a greedy policy with respect to V_k. It has been proved that by iterating the process, the estimation of the value function converges to the true value function if $\gamma < 1$. In this case, the greedy policy with respect to the estimation of the value function is an optimal policy. An in-depth discussion on bounds of for greedy policy with respect to value function in discrete MDPs is presented in [Williams and Baird 1994].

The approach is suited for problems with a finite number of states and actions. On continuous problems, it is impossible to iterate on all the states and actions. However, it is still possible to discretize a continuous problem in order to get a finite number of states and actions and then to solve the approximation of the original problem, see section . Another scheme is to use function approximators to store the values in the continuous space.

Policy iteration

Policy iteration algorithms also allow to obtain optimal policies for MDP, and are also based on choosing greedy action with respect to an estimated value function. The main difference is that instead of storing the last value function and updating it with a single step, they store the last policy and compute either its exact value or its approximated value after a large set of steps. The update step is based on Equation (7).

$$\pi_{k+1}(s) = \arg\max_{a \in A} \sum_{s' \in S} \int_{r \in \mathbb{R}} \delta(s', r, 1|s, a)\, r + \delta(s', r, 0|s, a)\,(r + \gamma V^{\pi_k}(s'))\, dr \quad (7)$$

Policy iteration algorithms require significantly less iterations than value iterations algorithms. However, their iterations require more computational power since they require to estimate the true value function of the current policy. An in depth-study of the performance bounds for value iteration and policy iteration is presented in [Scherrer 2013]. In this article, the author develop the theoretical study of the λ policy iteration algorithm, proposed as an unifying algorithm for both value and policy iteration. A thorough review of λ policy iteration and its implementations is presented in [Bertsekas 2013].

The initial form of policy iteration is not directly adaptable for continuous state and action spaces, because Equation (7) requires to find the optimal action for each state. While it is possible to approximate by experimenting all the couples in discrete spaces, enumerating them is not possible in continuous spaces.

Policy Iteration has inspired approaches such as policy gradient in which the shape of the policy is chosen using expert knowledge and its parameters are tuned during the learning process, see section .

Q-Value iteration

Q-learning algorithms are based on the Q-value. One of the main advantage of this approach is the fact that it does not require explicitly to build a transition model for the MDP. While building a model might lead to high sample efficiency, see [Deisenroth and Rasmussen 2011], it is not mandatory.

Successor, reward and terminal status

Often, the literature uses a deterministic reward function separated from the sampling of the next state, it also tend to consider that terminal status is directly a function of the state. While this formalism is suited to most of the problems, it leads to issues for some problems, specifically when the transition between state and actions is the result of a simulation with multiple steps.

Therefore, in this thesis we consider that the reward and the terminal status are sampled simultaneously with the successor.

In order to highlight the potential issues of separating reward, terminal status and successor, we refer to the *kicker robot problem*, see 1.3. In this problem a robot has to move on a soccer field in order to get closer to the ball and then shoot the ball. The goal of the agent is to reduce the average time required to score a goal. The state space is defined by the position and the orientation of the player and by the position of the ball. The action space includes choosing which kick the player can use as well as the direction of the kick. In order to be able to kick, the robot has to get close to the ball. This task is performed by simulating a *ball approach*, see section 1.2. Since the reward received at every step depend on the time required to reach the kick position, knowing the initial state, the action and the successor state does not provide enough information to have access to the reward. In this problem, there is several way to end up with a terminal status: the ball might be kicked outside of the field or there might be a simulated collision between the robot and the opponent goalie inside the penalty area. In the second case, the final position of the robot and the ball do not provide enough information to conclude if the state is terminal or not. Of course it would be possible to increase the state space to include a specific terminal state, but it is much simpler to have the transition function sample simultaneously the successor, the reward and the terminal status.

Heterogeneous action spaces

Usually, CSA-MDP uses a single type of action with continuous parameters. Those parameters might be the direction of the action, its intensity or any other continuous variables. However, in real-life situations robots might have to choose between semantically different actions which have entirely different parameters. Hybrid-MDP with both continuous and discrete variables are used in [Meuleau *and al.* 2009]. Although they allow to express both discrete and continuous aspects for states, they do not allow easily to model problems with several different types of actions. It is with this interest in mind that we chose to express the action space as a set of spaces. The *kicker robot* problem, see 1.3, highlights this advantage. With our framework, it is easy to specify different types of kick controller. While some actions might take only the direction as a parameter, other might require to specify the power as well.

Consider $A = \{\mathcal{H}_1, \ldots, \mathcal{H}_n\}$ an action space, it is an heterogeneous action space if and only if $\exists (i,j) \in \{1, \ldots, n\}^2, i \neq j \ \wedge \mathcal{H}_i \neq \mathcal{H}_j$

Learning in continuous spaces

Classical MDP approaches such as exact value iteration cannot be applied directly to CSA-MDP because there is an infinity of states. In this section we discuss the advantages and limits of different approaches used to compute policies for CSA-MDP.

Discretization

Approaches based on discretization convert CSA-MDP in MDP by gathering subspaces into a single state. This requires to partition the state and action spaces into a discrete, finite set of cells, it approximates the dynamics. This kind of process is often used to convert continuous problems into discrete problem.

Straightforward discretization has proven to be quite effective on low-dimensional problems, yielding satisfying results in [Benbrahim *and al.* 1992]. For higher dimension problems such as the ball-in-a-cup task in [Nemec *and al.* 2010], hand-crafted discretization require finer tuning but can still solve successfully the problem. Expert non uniform discretization of the state space has been proposed and discussed in [Santamaria *and al.* 1997].

While discretization tend to provide solutions quickly, it is inefficient in high-dimensional problems due to the curse of dimensionality.

Policy gradient

Policy gradient algorithms uses parametrized policies and models to solve CSA-MDP. They rely on optimization of the parameters through gradient descent. The gradient of the value function with respect to the parameters of the policy can be obtained through either sampling or analytic equations.

This approach has already been used successfully in robotics. Policy gradient algorithms have been able to learn control policies to hit a baseball with an anthropomorphic arm [Peters and Schaal 2006]. They can display an impressive sample efficiency [Deisenroth and Rasmussen 2011], making them suitable for robotic applications.

The constraints on access to a symbolic and derivable model of the transition and reward functions are difficult to satisfy in humanoid robotics due to the presence of collisions with the ground. Moreover, due to the large discrepancy between the model and the real-world experiments on our low-cost robots, using the analytic model for evaluating the robot dynamics is not realistic. Therefore, policy gradient algorithms are not suited for applications on low-cost robots which have a bad signal to noise ratio.

Local planning

Local planning algorithms do not solve a complete MDP but uses online information of the current state to strongly reduce the space which require exploration. They often uses a limited horizon in planning and therefore yield excellent performance on short-horizon tasks but they come at the price of a high online cost since the computations have to be performed online in order to have access to the current state. This point make it often impractical to use local planning in robotics, because the computational power is limited and decisions have to be taken in real-time.

In [Weinstein and Littman 2013], local planning algorithms are used to provide outstanding control of two-dimensional biped humanoids with seven degrees of freedom. However, each decision required 10'000 trajectories for a total of 130 seconds per time step on an Intel Core i7-3930k, a much more powerful CPU than those currently embedded on robots. Moreover, the amount of noise simulated is still much lower than the control and sensing noise on a real robot. Although those results are encouraging, they require major improvements in hardware to be applied.

AlphaGo is the first program able to win Go games against the best human players, see [Silver *and al.* 2016]. It is worth noting that in order to reach such a performance, the authors used local planning under the form of Monte Carlo Tree Search as well as offline learning under the form of deep neural networks.

Exact methods

A few years ago, Symbolic Dynamic Programming has been proposed to find exact solutions to CSA-MDP [Sanner *and al.* 2012]. This algorithm, based on eXtended Algebraic Decision Diagrams requires a symbolic model of both the transition function and the reward function. It also relies on several assumptions concerning the shape of these functions and is suited for a very short horizon. This result is not applicable for the presented applications because of the transition models and the horizon which should be considered.

Contributions

The main contributions of this thesis are:

- three new algorithms for solving CSA-MDPs,
- an in-depth experimental study of several robotic problems, leading to new insights on how to model MDPs in robotics,
- a generic MDP solver, released as a C++ project

The three algorithms we propose are respectively called *Fitted Policy Forest* (FPF), *Random Forests Policy Iteration* (RFPI) and *Policy Mutation Learner* (PML). These three algorithms are designed to be able to cope with real robotic problems, therefore they can address a large number of dimensions, and produce policies which can be used at a low computational time.

Fitted Policy Forest This algorithm relies on classical techniques to perform an estimation of the Q-value and then uses this information to extract a lightweight policy. The policy is represented compactly as a regression forest, thus it can be used to perform real-time decision-making. FPF can be used either in batch mode or in semi-online learning. This algorithm is presented and compared to state of the art algorithms in Section 2.2. This work was published in [Hofer and Gimbert 2016].

Random Forests Policy Iteration This algorithm performs a variant of policy iteration. The current policy is stored as a regression forest. Each improvement step consists in 1) approximate the value function using the current policy, 2) update the policy using the new value function. RFPI requires a blackbox model of a CSA-MDP. The algorithm is presented in 2.3 and is used to solve the ball approach problem in chapter 3. This work was published in [Hofer and Rouxel 2017].

Policy Mutation Learner This variant of policy iteration performs local modifications of a policy while ensuring that those local modifications do not impact negatively the global performance. This is the only algorithm presented here which can handle multiple heterogeneous action spaces. It is presented in 2.4 and its experimental results on the kicker robot problem are presented in chapter 4. This work is unpublished.

We focus on two robotic problems faced by the Rhoban Football Club during RoboCup games.

Ball approach In chapter 3, the ball approach problem is studied thoroughly. First, a predictive motion model is presented and data acquired from real-world experiments is used to tune up parameters. Second, policies are trained using the RFPI algorithm. Both simulation and real world experiments show that the policies produced by RFPI outperform those used to win the RoboCup in 2016.

Kicker robot In chapter 4, an in-depth analysis of the kicker robot problem is provided. Through experiments, we show how cooperation between robots can be used to reduce the average time required to score a goal.

We did implement all three algorithms and the environment needed to perform experiments, including Gazebo bindings, as a collection of open-source

C++ modules compatible with ROS (Robotic Operating System). These modules are structured in order to limit the number of dependencies and to ensure genericity. All the solvers and optimizers allow choosing the number of threads used in order to benefit from parallelism. The total number of lines written is around 34'000. More details on the source code are provided in Section 2.6.

Organization of the thesis

In this introduction we provide the basic notions used throughout the thesis. In chapter 1, we present the experimental problems we focused on. In chapter 2, we present our three algorithms and their performances. In chapter 3, we present the experimental results obtained with RFPI on the *ball approach problem*. In chapter 4, we present the experimental results obtained with PML on the *kicker robot problem*.

Chapter 1

Targeted problems

In this chapter, we present several problems used for experimentation in this thesis[1].

Section 1.1 introduces classical problems used for benchmark in continuous reinforcement learning. Section 1.2 presents the ball approach problem along with the training of a predictive motion model for our humanoid robot. Finally, section 1.3 introduces the kicker robot problem and discusses the incentives for robot cooperation in the RoboCup context.

1.1 Benchmark problems

In this section, we present problems widely used as benchmark for reinforcement algorithms. Those problems are used to benchmark the *Fitted Policy Forests* algorithm in section 2.2.

1.1.1 Double integrator

The double integrator, see Fig. 1.1, is a linear dynamics system where the aim of the controller is to reduce negative quadratic costs. The continuous state space consist of the position $p \in [-1, 1]$ and the velocity $v \in [-1, 1]$ of a car. The goal is to bring the car to an equilibrium state at $(p, v) = (0, 0)$ by controlling the acceleration $\alpha \in [-1, 1]$ of the car. There are two constraints: $|p| \leq 1$ and $|v| \leq 1$. In case any of the constraint is violated, a penalty of 50 is received and the experiment ends. In all other case, the cost of a state is $p^2 + a^2$. The control step used is 500[ms] and the integration step is 50[ms], the discount factor was set to $\gamma = 0.98$.

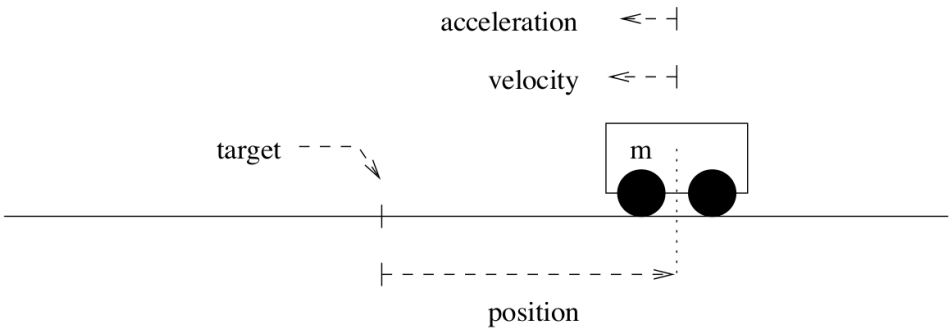


Figure 1.1 – The double integrator problem, from [Santamaria *and al.* 1997]

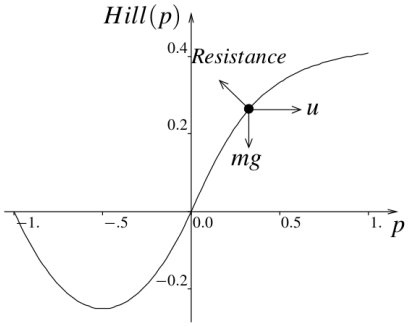


Figure 1.2 – The "car on the hill" problem, from [Ernst *and al.* 2005]

1.1.2 Car on the hill

In this problem an underactuated car must reach the top of a hill. space is composed of the position $p \in [-1, 1]$ and the speed $s \in [-3, 3]$ of the car while the action space is the acceleration of the car $u \in [-4, 4]$. The hill has a curved shape given by Eq. (1.1). If the car violate one of the two constraints: $p \geq -1$ and $|s| \leq 3$, it receives a negative reward of -1, if it reaches a state where $p > 1$ without breaking any constraint, it receive a reward of 1, in all other states, the reward is set to 0. The car need to move away from its target first in order to get momentum.

$$Hill(p) = \begin{cases} p^2 + p & \text{if } p < 0 \\ \frac{p}{\sqrt{1+5p^2}} & \text{if } p \geq 0 \end{cases} \tag{1.1}$$

[1] An open-source implementation in C++ of all the problem described here is available at: https://github.com/rhoban/csa_mdp_experiments.

1.1.3 Inverted pendulum stabilization

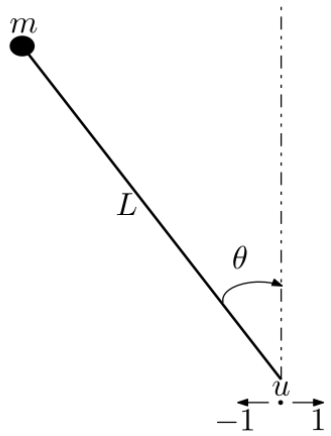


Figure 1.3 – The inverted pendulum stabilization problem, from [Bernstein and Shimkin 2010]

In this problem, the goal is to control the angular position of a pendulum linked to a cart through a free angular joint by applying a linear force f on the ground using the cart wheels, see Figure 1.3.

We use the description of the problem given in [Pazis and Lagoudakis 2009]. The state space is composed of the angular position of the pendulum θ and the angular speed of the pendulum $\dot{\theta}$, the action space is $[-50, 50]$ Newtons, an uniform noise in $[-10, 10]$ Newtons is added. The goal is to keep the pendulum perpendicular to the ground and the reward is formulated as following:

$$R(\theta, \dot{\theta}, f) = -\left((2\theta/\pi)^2 + \left(\dot{\theta}\right)^2 + \left(\frac{f}{50}\right)^2\right)$$

except if $|\theta| > \frac{\pi}{2}$, in this case the reward is -1000 and the state is considered as terminal. We set the discount rate γ to 0.95. The transitions of the system follow the nonlinear dynamics of the system described in [Wang *and al.* 1996]:

$$\ddot{\theta} = \frac{g\sin(\theta) - \alpha ml\left(\dot{\theta}\right)^2 \frac{\sin(2\theta)}{2} - \alpha\cos(\theta)u}{\frac{4l}{3} - \alpha ml\cos^2(\theta)}$$

where g is the constant of gravity $9.8[m/s^2]$, $m = 2.0[kg]$ is the mass of the pendulum, $M = 8.0[kg]$ is the mass of the cart, $l = 0.5[m]$ is the length of the pendulum, $\alpha = \frac{1}{m+M}$ and u is the final (noisy) action applied. We used a control step of $100[ms]$ and an integration step of $1[ms]$ (using Euler Method). The reward used in this description of the problem ensure that policies leading to a smoothness of motion and using low forces to balance the inverted pendulum are rated higher than others.

Figure 1.4 – An example of real-world ball approach

1.1.4 Inverted pendulum swing-up

In this problem, an angular joint with a pendulum is controlled directly and the aim is to stabilize it upward. The pendulum starts pointing downward and the torque of the angular joint is too low to lift it directly, therefore it is required to inject energy in the system by balancing the pendulum.

The main parameters are the following: the mass of the pendulum is $5[kg]$, the length of the pendulum is $1[m]$, the damping coefficient is $0.1[Nms/\text{rad}]$, the friction coefficient is $0.1[Nm]$, the maximal torque is $\tau_{\max} = 15[Nm]$, the maximal angular speed is $\dot{\theta}_{\max} = 10[\text{rad}/s]$ and the control frequency is $10[Hz]$. The reward function used is the following

$$r = -\left(\left\|\frac{\theta}{\pi}\right\| + \left(\frac{\tau}{\tau_{\max}}\right)^2\right) \tag{1.2}$$

Where θ is the angular position of the pendulum (0 denote an upward position), and τ represent the torque applied on the axis. If $\left\|\dot{\theta}\right\| > \dot{\theta}_{\max}$, a penalty of 50 is applied and the episode is terminated.

While the system only involves two state dimensions and one action dimension, it presents two main difficulties: first, random exploration is unlikely to produce samples where $\theta \approx 0$ and $\dot{\theta} \approx 0$ which is the target, second, it requires the use of the whole spectrum of actions, large actions in order to inject energy in the system and fine action in order to stabilize the system.

1.2 Ball approach

During robotic soccer games, several skills such as recovering from falls or kicking the ball are required, but most of the time is spent in the *approach* phase, see Figure 1.4. During the approach, the robot has an estimate of both the ball position and its own position on the field. It is then able to choose a desired aim for its next kick. The main problem is to control the orders sent to the walk engine in order to reach the desired position as fast as possible.

The walk predictive model and the observations gathered by the robots are stochastic. Due to the lack of predictability of the robot's motions, it is impossible to use standard methods for planning in continuous domains. Moreover, since the decisions have to be taken online and updated after every step, the choice of the orders should not require heavy computations.

1.2.1 Walk engine

The walk engine[2] used by the Sigmaban robot and presented in [Rouxel *and al.* 2016] is omni-directional and controls the target position of the 12 leg joints. It has been mainly developed by Grégoire Passault and Quentin Rouxel. The walk is externally driven by three values: the forward and lateral length of the next step and its angular rotation. These walk orders are issues at each complete walk cycle (two steps).

The engine is based on a set of parametrized polynomial splines in Cartesian space of the foot trajectories with respect to the robot's trunk. A stabilization procedure detects strong perturbations by looking for unexpected weight measures from the foot pressure sensors. In case of perturbations, the walk timing is altered in order to keep the robot balanced.

1.2.2 Almost non holonomic approach

While some humanoid robots are able to move very quickly laterally, other robots are mainly capable of moving forward, backward and rotate. For these robots, policies that rely on lateral motion are particularly slow. In order to test the flexibility of the policies, we designed two version of the problem: Holonomic Approach, HA for short and Almost Non Holonomic Approach, ANHA for short. In ANHA, we divided the limits for the lateral speed and acceleration by 5, since those robots are not efficient at moving laterally, we also divided by two the lateral noise with respect to HA.

1.2.3 State and action spaces

Sudden variations of the parameters provided to the walk engine can result in instability and fall of the robot. Therefore, we decided to control the acceleration of the robot and not the walk orders directly. The name, limits and units of the state space can be found at Table 1.1 and those of the action space at Table 1.2. We do not consider the ball speed in the state space for two reasons. First, it is very hard to get an appropriate estimate of the ball speed because the camera is moving with the robot and the position observations are already very noisy. Second, the ball is static most of the time, because robots spend a large proportion of their time trying to reach the ball.

Even with the restrictive bounds on speed, see Table 1.1, and acceleration, see Table 1.2. Some combinations of large speed and acceleration of walk orders can still make the robot unstable. These events trigger the stabilization procedure which manages to recover from the perturbations at the expense of high noise in the robot's displacement and extra-time allowed to the current step.

[2]Implementation is available at https://github.com/Rhoban/IKWalk

Table 1.1 – State space of the ball approach problem

		HA		ANHA	
Name	Units	min	max	min	max
Ball distance	m	0	1	0	1
Ball direction	rad	$-\pi$	π	$-\pi$	π
Kick direction	rad	$-\pi$	π	$-\pi$	π
Forward speed	$\frac{m}{\text{step}}$	-0.02	0.04	-0.02	0.04
Lateral speed	$\frac{m}{\text{step}}$	-0.02	0.02	-0.004	0.004
Angular speed	$\frac{\text{rad}}{\text{step}}$	-0.2	0.2	-0.2	0.2

Table 1.2 – Action space of the ball approach problem

		HA		ANHA	
Name	Units	min	max	min	max
Forward acceleration	$\frac{m}{\text{step}^2}$	-0.02	0.02	-0.02	0.02
Lateral acceleration	$\frac{m}{\text{step}^2}$	-0.01	0.01	-0.002	0.002
Angular acceleration	$\frac{\text{rad}}{\text{step}^2}$	-0.15	0.15	-0.15	0.15

1.2.4 Specific areas

At every step, the robot is given a reward which depends on the area it is located in:

Kick: The distance to the ball along the X-axis has to be between 0.15 m and 0.3 m. The absolute position of the ball along the Y-axis has to be lower than 0.06 m. The absolute angle between the robot direction and the kick target has to be lower than 10 degrees. Inside this area, the robot receives a reward of 0.

Collision: The position of the ball along the X-axis has to be between -0.20 m and 0.15 m. The absolute position of the ball along Y-axis has to be lower than 0.25 m. Inside this area, the robot receives a reward of -3.

Failure: The distance to the ball has to be higher than 1 m. In this case, we consider that the robot has failed the experiment, the state is terminal and the reward is -100.

Normal: For every state that does not belong to any of the previous cases. Inside this area, the robot receives a reward of -1 (i.e. unit cost).

This binary separation between the different areas rises a specific concern about continuity of the reward. While several learners use assumptions on the

Lipschitz constant of reward and transition functions in order to speed-up the process or to provide guarantees regarding execution time, in this problem, the reward function is not even continuous. Moreover, the reward is constant inside each area, thus leading to large plateau with discontinuities.

A crucial aspect to account for when evaluating the difficulty of a problem is the general shape of transition and reward functions. Discontinuities, large plateau and multiple local maxima are as important regarding the complexity of a problem as the number of dimensions. Multi-Resolution Exploration [Nouri and Littman 2009] provide performance guarantees with respect to the Lipschitz constants of the reward and the transition function. Bayesian optimization in [Brochu *and al.* 2010] assume that the function to optimize is Lipschitz-continuous even if knowledge of the value of the Lipschitz constant is not required.

The last challenging element for this problem is the fact that the *Kick* area is placed near to the *Collision* area. In order to optimize its policy, the solver needs to navigate nearby the *Collision* zone which has a strong penalty. This may lead to locally optimal policies consisting of staying away from both, the *Kick* and the *Collision* areas.

1.2.5 Initial state distribution

For the ball approach problem, the initial distance to the ball is sampled from an uniform distribution between 0.4 m and 0.95 m. The initial direction of the ball and the kick direction are sampled from an uniform distribution in $[-\pi, \pi]$. The initial velocity of the robot is always 0 (for both, Cartesian and angular velocities).

1.2.6 Calibration of the predictive motion model

One of the major disadvantages of using low-cost robots is their important mechanical and control inaccuracies. A large discrepancy can be observed between the orders provided and actual physical displacement of the robot. Despite this error, the deterministic part of the robot real behavior can still be captured and used to improve the control properties. An approach to capture the deterministic part of the error is presented here. The calibration of the predictive motion model is part of the work presented in [Hofer and Rouxel 2017].

First, we present the interests of training a model based on real data. Then we present the predictive motion model we use. Finally, we describe the experimental setup and results used to train the predictive motion model.

Training a surrogate model

Surrogate models are an approximation of another model where low computational complexity is obtained at the cost of accuracy. They are widely used when optimizing an expensive blackbox function. In our case, a surrogate model named predictive motion model is built from data because optimizing the policy online would require to gather a large number of real world samples, accepting the necessity of constant human supervision and the risk of damaging the robot. On the other hand, the discrepancy between the orders provided to the walk engine and the real motion is too large to be simply ignored.

A major interest of using surrogate models for real world experiments is the fact that the initial state of the robot cannot be chosen freely in the real world. Therefore, surrogate models are more convenient than reality because they allow choosing the initial state and the action without requiring any setup.

In our case, the surrogate model is trained using data acquired on the real robot. In order to avoid overfitting, we base our evaluation of the models on cross-validation. This method of validation separates the data in two sets: the training set and the validation set. The optimization of the blackbox function is performed on the training set while the evaluation of the performance uses only the validation set. An extensive study on the interests and benefits of cross-validation is presented in [Kohavi 1995].

Predictive motion model

Calibration of online odometry and predictive motion model have been previously studied in [Rouxel *and al.* 2016]. The odometry evaluates during the robot's motion its own relative displacement by analyzing the sensors readings. On the contrary, the predictive motion model tries to predict the future robot self displacements given a sequence of orders sent to the walk engine.

While the work presented in [Rouxel *and al.* 2016] used a motion capture setup to learn a non linear, non parametric regression model, only simple linear models are considered in this thesis. Since they have a much smaller number of parameters, they require less samples for training and therefore do not require to deploy a motion capture setup at the RoboCup competitions. The method proposed in this section require only a measuring tape and is much more convenient to use outside of the laboratory.

The linear model we use is based on Equation (1.3). The parameters of the model are the $a_{0,0}...a_{2,3}$ coefficients.

$$\begin{bmatrix} \Delta x_{corrected} \\ \Delta y_{corrected} \\ \Delta \theta_{corrected} \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \begin{bmatrix} 1 \\ \Delta x \\ \Delta y \\ \Delta \theta \end{bmatrix} \tag{1.3}$$

The surrogate model of the walk engine include an amount of noise on

position and orientation at each step, this reflects the stochastic aspect of the walk. We considered noise drawn uniformly from the hyperrectangle $\mathcal{H}_{\text{stepNoise}}$ with:

$$\mathcal{H}_{\text{stepNoise}} = \begin{bmatrix} -0.02 & 0.02 \\ -0.02 & 0.02 \\ -\frac{5\pi}{180} & \frac{5\pi}{180} \end{bmatrix}$$

In

Algorithm 1 Multiple steps predictive model

function GETFINALPOSITION(A,orders,applyNoise)
▷ Parameters description:

A	The matrix of coefficients of the model
orders	The list of orders to apply
noise	A boolean value indicating if prediction includes noise

$(x, y, \theta) = (0, 0, 0)$
for all $(\Delta x, \Delta y, \Delta\theta) \in$ orders **do**
$\begin{bmatrix}\Delta x & \Delta y & \Delta\theta\end{bmatrix}^T = A \begin{bmatrix}1 & \Delta x & \Delta y & \Delta\theta\end{bmatrix}^T$ ▷ see Eq 1.3
if noise **then**
$\begin{bmatrix}\Delta x & \Delta y & \Delta\theta\end{bmatrix}^T += \text{sample}\left(\mathcal{U}\left(\mathcal{H}_{\text{stepNoise}}\right)\right)$
end if
$x = x + \cos(\theta)\Delta x - \sin(\theta)\Delta y$
$y = y + \sin(\theta)\Delta x + \cos(\theta)\Delta y$
$\theta = x + \Delta\theta$
end for
return $\begin{bmatrix}x & y & \theta\end{bmatrix}^T$
end function

We consider three different types of model based on Equation 1.3:

Proportional	Three coefficients are available $a_{0,1}, a_{1,2}, a_{2,3}$
Linear	Six coefficients are available $a_{0,0}, a_{1,0}, a_{2,0}, a_{0,1}, a_{1,2}, a_{2,3}$
Full	All the 12 coefficients of A are available

Data acquisition

To generate learning data, the following procedure is used to generate a sequence:

- Minimum and maximum bounds are set according with the problem on both raw walk orders (velocity) and delta walk orders (acceleration).
- The robot is placed at a known starting position and orientation.

- During 15 seconds, random orders are sent at each step to the walk engine. The orders are drawn from a random walk (Brownian motion) where accelerations are uniformly distributed inside allowed ranges. All the orders sent to the walk engine are stored as part of the data.

- Final position is manually measured with respect to starting position. Since final orientation is difficult to measure accurately, it is only recorded for 12 possible orientations (more or less 30 degrees).

Consider a sequence seq, the orders sent to the walk engine are stored from the first applied to the last and are noted orders(seq). The measurement of the final position is noted observation(seq).

This process was repeated 25 times to gather enough data to approximate the predictive model. We consider the data stored as a list of sequences.

Measurements of the final position and orientation are subject to an important noise since the are performed manually. Moreover, each measurement takes between 15 and 30 seconds. By sending random orders during 15 seconds rather than measuring the position after every step, we divide the impact of measurement noise and increase the ratio of data time over measurement time.

Training procedure

Optimization of the parameters of predictive model is performed using the CMA-ES algorithm which is described in Appendix A.1. The cost function to optimize is deterministic and described in Algorithm 2. Optimization of cost functions is discussed further in section 2.4.1.

Algorithm 2 The cost function for predictive model training

function GETTRAININGREWARD(A,S)
  ▷ Parameters description:
    A The matrix of coefficients of the model
    S The set of training sequences
  totalError $= 0$
  for all seq $\in S$ **do**
    predicted $=$ getFinalPosition(A, orders(seq), False)
    measured $=$ observation(seq)
    totalError$+=$ squaredError(predicted, measured)
  end for
  return $\frac{\sqrt{\text{totalError}}}{|S|}$
end function

Optimization results

The figure 1.5 displays the results of the motion prediction models calibration process. It shows the prediction error on the cross-validation set with confidence bound as a function of the number of sequences used for training. The proportional model strongly outperforms the simple linear model with a small training set, and stands similar performances for the largest training set. The main reason for this fact is that the walk engine had been tuned to result in a steady walk when no orders are provided. Therefore, the simple linear model has more parameters to tune while the gain in expressiveness is not really useful. On the other hand, the full linear model outperforms other models once the number of sequences available is above 15. The advantage of using a full linear model is due to the fact that order along a dimension have effects on the other dimensions, see figure 1.7.

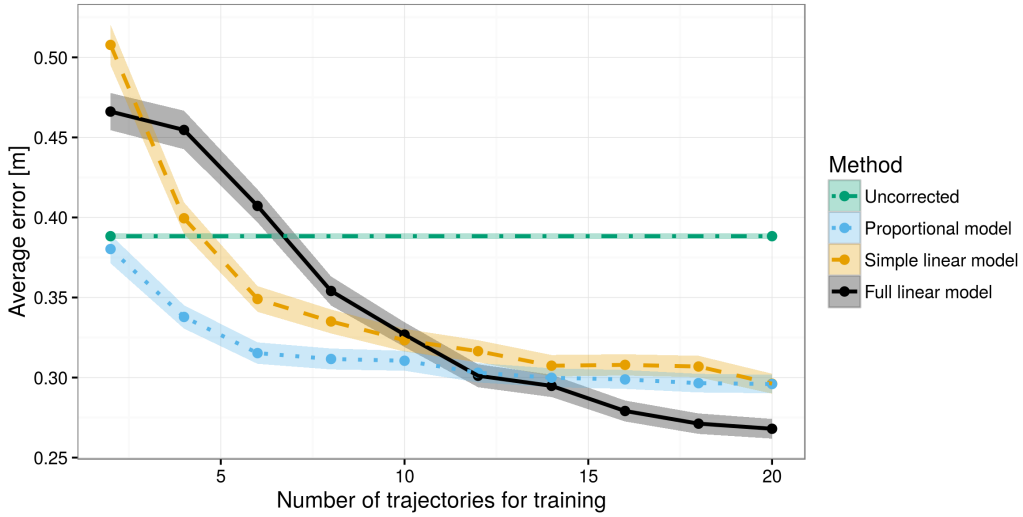


Figure 1.5 – Performance of motion predictive models on cross-validation set

While cross-validation is a satisfying test against the risk of overfitting, it is still possible that several set of parameters provide acceptable results for the model. Ensuring that the learning process produces similar sets of parameters for various training sets helps ensuring the reliability of the learned model. Figure 1.6 and figure 1.7 present evolution of the confidence intervals for the parameters depending on the number of sequences used as part of the training set.

1.3 Kicker robot

In the *kicker robot problem*, one or several players of the same team try to score a goal as quickly as possible against a static goalie. This problem occurs

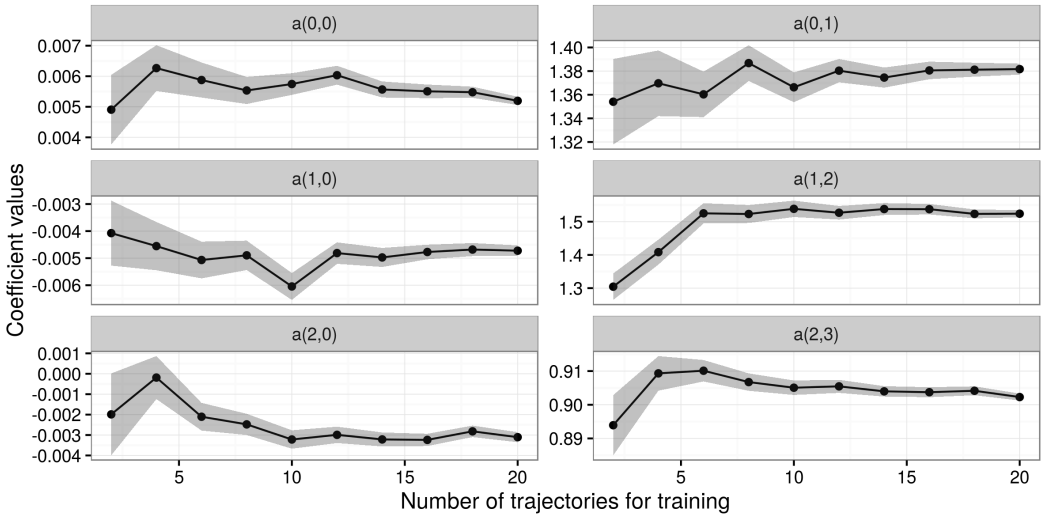


Figure 1.6 – Convergence of parameters for the simple linear model

frequently at RoboCup. The control of the walk engine is ensured by predefined policies and the decision which has to be taken by the robots consist of choosing three properties regarding the kick: the robot performing the kick, the type of the kick and the direction of the kick. In such situations, the robots may cooperate, typically a defender may pass to the attacker in order to score a goal as quickly as possible. An example of cooperation between two robots is given in Figure 1.8.

Historically, the first strategies used by the team in such situations were suffering several drawbacks. The strategy did consist in a rudimentary hand-made geometrical approach: the closest player kicked toward the center of the goal with the maximal power. There are three drawbacks with such a strategy. First, when the ball is near the goal line, the wished direction of the kick is almost parallel to the goal line. Second, geometrical approaches do not handle the noise on the kick direction, therefore, some kicks might have a very low probability of scoring because the acceptable tolerance on direction is low. Finally, including the possibility of performing a pass in this kind of strategies is difficult.

The reward function of the *kicker robot problem* is based on the time required to score a goal. Penalties applies if the ball collides against the opponent goalie or get out of the field. The reward function is discussed in details in section 1.3.3.

In the description of the *kicker robot problem*, we discuss three important parameters the problem. First, there can be one or two players, this affects both state space and action space. Second, the set of available actions can vary depending on robots low-level abilities. Third, different methods can be used to simulate the motion of the robots. Those three aspects are introduced

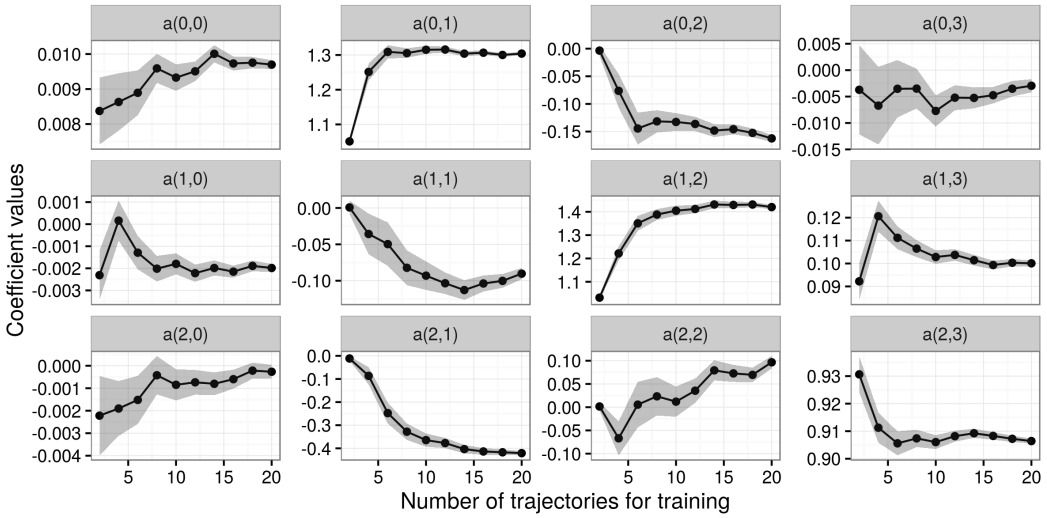


Figure 1.7 – Convergence of parameters for the full linear model

Figure 1.8 – Real-world cooperation

and discussed in this section.

1.3.1 State space

A state of the *kicker robot problem* is defined by the position of the ball and the position and orientation of the robots. Therefore, if there is only 1 robot, the dimension of the state space is 5 and if there are two robots the dimension of the state space is 8. The limits of the state space are defined by the size of the field. In this thesis, we consider 8 meters long and 6 meters wide field. The ball cannot exit those bounds: as soon as the ball crosses the boundaries, the game is over (i.e. the transition has a terminal status. The orientation of the robots is bounded in $[-\pi, \pi]$.

The initial state of the *kicker robot problem* is set according to the following process. First, the position of the ball is sampled randomly on the field, with a uniform distribution. Then, the initial state of each robot is sampled randomly inside the field. If a robot is further than 4 meters of the ball, then the sampling of its initial state is repeated until a position close enough is sampled.

1.3.2 Kick actions

Kick motions

Performing a kick with a humanoid robot requires the ability to balance on one foot while moving the other leg dynamically. Since we use low-cost motors, the amount of noise in action and perception is too important to simply apply the equations from the rigid body dynamics. Therefore we mainly rely on hand-tuning of splines to achieve stable and robust kick motions. In order to avoid spending too much time on hand-tuning tasks, we decided to have only three different kick motions.

Powerful kick The robot shoots at around 3 meters in front of him, an example is shown in Fig. 1.9.

Small kick The robot shoots at around 1.5 meters in front of him. The motion is similar to the *powerful* kick, but slower.

Lateral kick The robot pushes the ball on the side using the inner part of the foot. The average distance traveled by the ball is around 2 meters. An example of lateral kick is shown in Fig. 1.10.

Figure 1.9 – A *powerful* kick

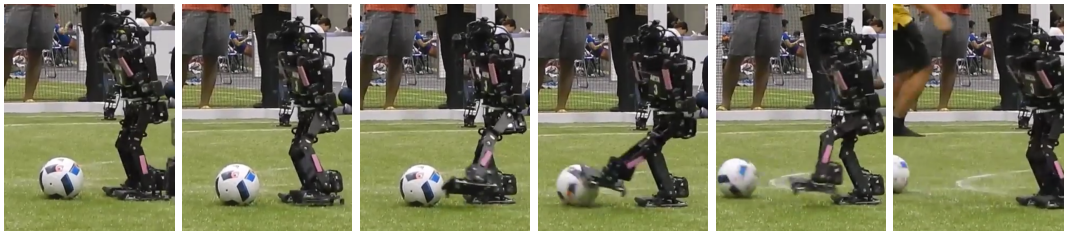


Figure 1.10 – A *lateral* kick

Kick decision models

Aside from choosing the type of kick, the robots have to choose the direction of the kick. While the simplest version would be to represent the desired direction as an angle in $[-\pi, \pi]$, this leads to complex models for expressing the simple strategy of kicking always to the center of the goal. Therefore we decided to have two different kick decision models.

Classic The continuous parameter is the direction of the kick θ in the field referential, with $\theta \in [-\pi, \pi]$

Finisher This kick uses a symbolic target on the field to compute the desired direction. The continuous parameter is the desired value for the y coordinate of the ball y_{wished} when crossing the goal line, with $y_{\text{wished}} \in [-\frac{\text{goalWidth}}{2}, \frac{\text{goalWidth}}{2}]$.

Actions

We present here the list of available actions. The first word identifies the kick decision model and the second one identifies the kick motion.

1. Finisher powerful
2. Classic small
3. Classic powerful
4. Finisher lateral
5. Classic lateral

We distinguish two types of problems: first, *classic problem*, where only actions 1 to 3 are allowed, second, *lateral problem*, where all the actions are allowed. For 2 robots problem, the set of actions is duplicated, since both robots can perform the same kicks.

Through this proposition, expressing the simple strategy of aiming towards the center of the goal and kicking with the maximal power can now be expressed using a simple constant model in which the action is always *Finisher Powerful* with a constant parameter 0.

1.3.3 Transition function

The transition process is divided in several phases, presented briefly here and discussed with more details after.

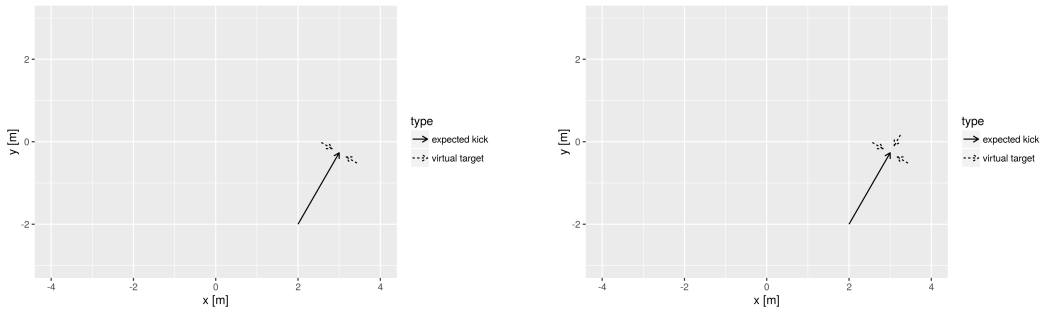
1. The ball is moved randomly using a noise sampled from a uniform distribution: $\mathcal{U}\left(\begin{bmatrix} -0.1 & 0.1 \\ -0.1 & 0.1 \end{bmatrix}\right)$. This noise represents the perception noise, it has a major impact when the ball is nearby the opponent goal.

2. The kicker moves towards the ball. For problems with two robots, the other robot moves towards a virtual target defined by the expected kick.

3. The kick is performed by the kicker. For problems with two robots, the other robots has a time budget of 10 seconds to move during the other robots perform the kick.

Virtual targets for non-kicking robots

When there are two robots playing in cooperation, a virtual target is provided to the robot which is not performing the kick. There are two potential targets available based on the expected trajectory for the kick, or even three if lateral kicks are available. Examples of the allowed targets are provided in Fig. 1.11.



(a) Lateral kick forbidden (b) Lateral kick authorized

Figure 1.11 – Virtual targets for non-kicker robots

Consider $\theta_{\text{kick}} \in \begin{bmatrix} -\pi & \pi \end{bmatrix}$ the direction of the kick in the field referential. We define the set of target orientation Θ by:

$$\Theta = \begin{cases} \left\{\theta_{\text{kick}} - \frac{\pi}{2}, \theta_{\text{kick}} + \frac{\pi}{2}\right\} & \text{if lateral kicks are forbidden} \\ \left\{\theta_{\text{kick}} - \frac{\pi}{2}, \theta_{\text{kick}} + \frac{\pi}{2}, \theta_{\text{kick}} + \pi\right\} & \text{if lateral kicks are authorized} \end{cases}$$

Given a set of target orientations Θ and $(x, y) \in \mathbb{R}^2$ the ball expected position after the kick, we define the set of allowed targets T as follows:

$$T = \{(x + cos(-\theta), y + sin(-\theta), \theta) \mid \theta \in \Theta\}$$

When moving, the non-kicking robot always selects the closest target among allowed targets.

Robots motions

The most accurate way at our disposal to simulate the motion of the robots is to use the motion predictive model presented in Section 1.2.6. However, since the robot can walk around 100 steps between two kicks, this type of simulation is computationally expensive with respect to simple approximations such as computing the distance and dividing it by a rough estimation of the speed.

There is a difference between the robot performing the kick and the other robot. The robot performing the kick is used to determine the time needed to perform the approach. The motion of the other robot is based on a time budget, defined either by the time required for the kicking robot to reach its target (approach phase) or by the time required to perform a kick (kick phase).

In order to experiment the trade-off between prediction accuracy and computational complexity, we propose 2 different types of approaches:

Simul Simulation of the real process, each step is taken successively, the approach is simulated and noise is added at each step. The position of the kicking robot is tested after each step to see if it can perform the requested kick. The number of steps performed by the non-kicking robot is based on its time budget.

Speed Simplest way of estimating the time required to reach the kick position. The distance is computed and divided by a rough estimation of the Cartesian speed, 10[cm/s] and the angular speed $\pi/4$[rad/s]. When a robot is kicking, the speeds are used to compute the time requested to reach the position. If a robot is not kicking, it starts by moving using Cartesian motion, if it reaches the target position within the time budget, it starts rotating.

Reward and terminal states

The *kicker robot problem* uses time to score a goal as a basis for its cost function. Therefore, all rewards are negative. During the approach phases time spent for approach is sampled based on the motion model. Each kicking action costs an additional 10 seconds, representing the time required for both stabilization and motion.

If the ball leaves the field or touches the goalie, the status of the transition is terminal. Moreover, a reward of -50 is added if the ball touches the goalie and a reward of -500 is added if the ball leaves the field outside of the goal.

Applying kicks

We distinguish the noise on the kick direction in two different components: first, the real orientation of the robot when kicking the ball which is not exactly the desired orientation, second, the direction of the ball in the robot referential.

The real direction of the kick is sampled according to Eq. (1.4), with the following parameters:

θ_{wished} The direction the robot aims to kick in the field referential.

θ_{real} The real direction taken by the ball after the shoot in the field referential.

n_{approach} The noise involved in approach and localization combined, the chosen value is 5 degrees.

σ_{kickDir} The standard deviation of the kick direction with respect to the mean direction. A standard deviation of 10 degrees is used for *small* and *powerful* kick while *lateral* kick has 15 degrees of standard deviation. Those values were approximated roughly during real world experiments at the RoboCup 2017.

$$\theta_{\text{real}} = \theta_{\text{wished}} + \mathcal{U}\left(\begin{bmatrix} -n_{\text{approach}} & n_{\text{approach}} \end{bmatrix}\right) + \mathcal{N}(0, \sigma_{\text{kickDir}}) \tag{1.4}$$

Once the direction of the kick has been computed, it is still necessary to compute the distance traveled by the ball. Through our experiments, we measured that the noise on the distance traveled was higher for powerful shoots. Therefore, we implemented the computation of the traveled distance with the following equation: $d_{\text{real}} = \mathcal{N}(1, \sigma_{\text{kickPower}})d_{\text{kick}}$ with:

d_{real} The final distance traveled by the ball

d_{kick} The nominal distance for the chosen kick

$\sigma_{\text{kickPower}}$ The standard deviation on the factor multiplying the nominal distance of the kick. For *powerful* and *lateral* kicks, the value was 0.2, while for *small* kicks it was 0.1.

1.3.4 On cooperation between robots

While cooperation between players has always been a central issue in human soccer, its presence at the RoboCup competition strongly depends on the leagues. In simulation leagues as well as wheeled robots leagues, team-play is essential to be able to score a goal. On the other side, for humanoid leagues, the problem is quite different. In adult size, there is only one robot per team, mainly because of the price and amount of engineering involved in building a humanoid-size robot. In standard platform league, where all robots are Nao, the best teams have been able to display cooperation behavior in order to avoid opponents. However, in kid-size and teen-size, very few teams have been able to perform passes between robots.

Among the most important barrier to the development of team-play in the KidSize league, the most obvious one is the high impact of noise on the final position of the ball. With the recent introduction of artificial turf for the field in 2015, predicting the outcome of kick has become more difficult. Due to the inclination of the blade of the grass, the average distance traveled by the ball strongly depends on the direction of the kick. During the RoboCup 2017, we have observed variations from 1.5 meters to 3 meters depending on the direction of the kick. Under those circumstances, we were unable to perform passes before taking into account a basic model of the grass in our predictions.

As discussed in chapter 3, preparing a kick by accurate positioning is a difficult task for humanoid robots and takes a significant amount of time. Thus, the primary objective is to minimize the average number of kicks required by the robots and the distance walked by the robots is often secondary. We expect that improving the quality of the control in the last few steps can reduce the cost of approaching the ball and therefore make it more interesting to use several passes to score a goal faster. Currently, it is often more interesting for a robot to kick the ball toward the opposite goal and to follow it after than to try passing the ball to a teammate.

One of the main incentive for team-play in soccer is to reduce the risk of opponents catching the ball. However, detecting other robots is particularly difficult in the KidSize league due to heterogeneous nature of robots. In the SPL league, all the robots are Nao with different colors of jersey, therefore teams can train classifiers to detect robots using their own robot. Since developing a robust self-localization system is already difficult, there is generally not much computational power left for opponent detection. Since we do not have access to the opponent position, we did not include eventual opponents in the state space, except for the goalie.

Despite all of these problems, we show in chapter 4 that even with an important noise on the kick results and a lack of information on the opponent position, cooperation between robots is still interesting.

Chapter 2

Computing efficient policies

In this chapter, we present the algorithms proposed during this thesis. Section 2.1 introduces the notions of regression trees and regression forests along with operations on those elements. Section 2.2 presents the *fitted policy forests* algorithm and results it obtained on classical benchmark problems. Section 2.3 presents the *random forest policy iteration algorithm*. Section 2.4 starts by presenting the concept of blackbox optimization and then introduces the *policy mutation learner* algorithm. Finally, section 2.6 presents the open-source contributions produced during this thesis.

2.1 Tree-based regression

While it is possible to store an exact model of discrete MDPs with a finite number of states, working in continuous domains requires either to manipulate symbolic functions or to approximate functions representing the Q-value, the value, the transition function or the policies. For that, we use function approximators.

In this section, we start by defining function approximators, then we introduce regression trees and regression forests and present the ExtraTrees algorithm, allowing to grow a regression forest from samples. Finally we present some algorithms performing basic operations on regression trees: projection, averaging and pruning.

2.1.1 Function approximators

A function approximator[1] allows to predict the output value $y \in Y$ corresponding to an input value $x \in X$, with $X \in \mathbb{R}^n$ and $Y \in \mathbb{R}^n$. We note $\tilde{f}$ the function approximator for function $f : X \mapsto Y$.

[1] An open-source C++ implementation of all the function approximators used in this thesis is available at https://github.com/rhoban/rosban_fa

Function approximators are typically used in learning for regression. In this case, we use a set of observed data to train a model, thus allowing prediction of the output for other inputs. Note that if the output is discrete, this problem is known as classification. Regression might be used as a basis for decision making in artificial intelligence, but it can also be used to build 'understandable' models of complex systems in order to identify the most important dimensions of the input and their impact on the measured output.

Another use of function approximators is to save computation time by building a simplified model of the function which can easily be computed, typically in constant time. One of the most common examples for this use case is the sinus tables which can be pre-computed in order to save time during execution. The speed-up achieved through use of function approximators is even more obvious when we need to predict the average of a stochastic function. This can generally be achieved by Monte-Carlo methods involving a large number of rollouts. The cost of using a large number of simulations might often be prohibitive in real-time situations. Therefore, computing function approximators offline and using them online might significantly reduce the time consumption. However, using function approximators might lead to an unacceptable loss of precision, especially when dealing high-dimensional spaces or when the density of samples is low with respect to the variation of output in the area.

While most of the function approximators allows to map a multi-dimensional input to an mono-dimensional output, it is trivial to use them to predict multi-dimensional outputs. Let $f : X \mapsto Y$ be the function to approximate and $k = \dim_R(Y)$, then we can build a predictor $\tilde{f} : X \mapsto Y$ based only on k mono-dimensional output approximators $\tilde{f}_1, \ldots, \tilde{f}_k$. Simply using an approximator for each output dimension is enough: $\tilde{f}(x) = \left(\tilde{f}_1(x), \ldots, \tilde{f}_k(x)\right)$. More details about multi-output regression can be found in [Borchani *and al.* 2015], where various schemes are proposed to handle dependent output dimensions.

In this thesis we rely on four types of function approximators. First, basic models such as the constant and linear models. Second, some more elaborate data structure known as *regression trees* and *regression forests*.

2.1.2 Basic models

A *constant model* is a function approximator with constant output. It is noted $\mathcal{C}(y)$ with $y \in Y$ the output of the model.

A *linear model* is a function approximator with an output varying linearly depending on the input value. A linear model from space $X \in \mathbb{R}^n$ to $Y \in \mathbb{R}^n$ is noted $\mathcal{L}(A, B)$, with A a $\dim_R(Y) \times \dim_R(X)$ matrix and B a vector of length $\dim_R(Y)$. Consider a linear model $l = \mathcal{L}(A, B)$, the prediction for an input $x \in X$ is defined by $l(x) = Ax + B$.

2.1.3 Regression trees

A regression tree is a tree whose every inner node has degree two and is associated with a *split* and every leaf is associated with a basic function approximator, typically a linear model.

An example of a regression tree is shown in Figure 2.1. Formally, an *univariate split* or, simply, a *split* is a couple (d, v) with $d \in \mathbb{N}$ the dimension of the split and $v \in \mathbb{R}$ the value of the split.

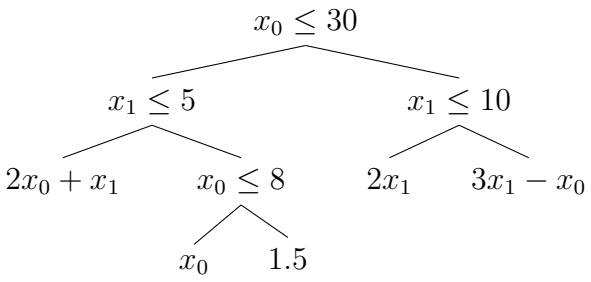


Figure 2.1 – A simple regression tree with linear approximators in leaves

Remark that there are more general notions of regression trees, for example the function approximators in the leaf may be polynomial functions and the splits can use more general inequalities for example linear splits such as $x_0 + 3x_1 \leq 4$. However in this thesis we focus on the simple and robust model of regression tress with univariate splits and constant or linear function approximators.

One of the main advantages of univariate splits is that they separate an hyperrectangle in two hyperrectangles. Therefore, it is particularly easy to sample elements inside the leaves of a regression tree if we limit input spaces to hyperrectangles and splits to univariate splits. In the sequel, given an hyperrectangle $\mathcal{H}$, a split dimension d and a value v, we denote:

$$\begin{aligned} \textsc{lowerSpace}(\mathcal{H}, d, v) &= \{x \in \mathcal{H} \mid x_d \leq v\} \\ \textsc{upperSpace}(\mathcal{H}, d, v) &= \{x \in \mathcal{H} \mid x_d > v\} \ . \end{aligned}$$

From now on, we will refer to splits by univariate splits, unless explicitly stated otherwise.

We denote a regression tree by $t = \mathcal{T}\left(d, v, \tilde{f}_1, \tilde{f}_2\right)$ where d is the dimension of the split, v its value and both $\tilde{f}_1$ and $\tilde{f}_2$ are either regression trees or constant or linear models. Of course we require a regression tree to be finite.

Consider a regression tree $t = \mathcal{T}\left(d, v, \tilde{f}_1, \tilde{f}_2\right)$. The value of t for an input

$x \in \mathbb{R}^n$ is denoted $t(x)$ and given by:

$$t(x) = \begin{cases} \tilde{f}_1(x), & \text{if } x_d < v \\ \tilde{f}_2(x), & \text{otherwise} \end{cases} \tag{2.1}$$

The dimension of the split is d and is noted $\textsc{splitDim}(t)$. The value of the split is v and is noted $\textsc{splitVal}(t)$. The lower child of t is $\tilde{f}_1$ and is noted $\mathrm{LC}(t)$. The upper child of t is $\tilde{f}_2$ is noted $\mathrm{UC}(t)$. The *domain* of t is the hyperrectangle $\mathcal{H}(t)$ defined by

$$\mathcal{H}(t) = \begin{cases} \mathbb{R}^n & \text{if } t \text{ has no ancestor,} \\ \textsc{lowerSpace}(\mathcal{H}(t'), d, v) & \text{if } t = \mathrm{LC}(t') \\ \textsc{upperSpace}(\mathcal{H}(t'), d, v) & \text{if } t = \mathrm{UC}(t') \ . \end{cases}$$

We denote $\textsc{isLeaf}$ the function checking whether a node of a regression tree is a leaf. These notions are illustrated by Algorithm 3, which computes the number of leaves of a regression tree.

Algorithm 3 Counting the leaves of a regression tree

```
1: function nbLeaves(t)                          ▷ t: a regression tree
2:     n ← 0
3:     for all f̃ ∈ {LC(t), UC(t)} do
4:         if isLeaf(f̃) then
5:             n ← n + 1
6:         else
7:             n ← n + nbLeaves(f̃)
8:         end if
9:     end for
10:    return n
11: end function
```

History

Originally, regression trees have been widely developed and used in experimental science as a statistical tool allowing to highlight statistical evidences of the effect of variables on the output. They present the huge advantage of being easily understandable by humans since the most impacting variables are positioned near the top of the tree, see [Breiman *and al.* 1984]. A side-effect of this origin is the fact that most of the algorithms tend to be too conservative, thus ending up with poor prediction performance with respects to methods which uses cross-validation to avoid over-fitting.

Growing regression trees

While the internal content of the algorithms allowing to build regression trees can change, they tend to share a common structure, typical of tree search. First, they start at the root node, then they optimize the choice of the split for the current node and repeat the process for all children of the current node until a stopping criteria is reached. Finally, they might use a pruning scheme to reduce the size of the tree and the risk of overfitting. A major issue with this general scheme is the fact that only immediate gain of splits is considered. Therefore, there is no planning and selection of the best combination of splits to fit the data available due to the computational complexity, see [Loh 2011].

2.1.4 Regression forests

Regression forests are based on the idea of building multiple regression trees and aggregating their prediction to provide a more reliable prediction. A regression forest $f = \mathcal{F}(t_1, \ldots, t_n)$, is a set of regression trees which can be used as a function approximator by averaging, using equation 2.2.

$$\mathcal{F}(x) = \frac{\sum_{i=1}^{n} t_i(x)}{n} \tag{2.2}$$

Consider a regression forest $f = \mathcal{F}(t_1, \ldots, t_n)$, the number of trees in f is denoted NBTREES(f). The i-th tree of f is denoted GETTREE(f, i).

The approach of producing multiple predictors from a single data set and using them to get an aggregated predictor has first been proposed in [Breiman 1996]. He proposes to build k replicates of the data set, drawn at random with replacement from the initial data set. Each of this k different data sets is then used to train a different predictor.

When predicting, the method used for aggregating is different depending on the output form. For classification problems, a plurality vote is taken and for regression problems, the average is chosen. This procedure is called bootstrap aggregating, or *bagging* for short. Improvement on the prediction using bagging is particularly significant for algorithms where small changes in the data set might result on large changes in prediction. Therefore, it is particularly suited for regression trees, because small changes in the data set might influence the dimension used for splits. Once a split has changed, the structure of the new subtrees might be very different of the structure of the old subtrees, thus leading to large changes in prediction.

There are two different approaches to use regression forest: bagging can be used to produce different data sets or the process used for growing trees might be stochastic. This second approach is explored in [Geurts *and al.* 2006] where extremely randomized trees are presented, ExtraTrees for short. The

ExtraTrees approach strongly reduces the computational time used to ensure that best splits are chosen, it includes randomness in both the choice of the split dimension and the value used to split the current space. While this method strongly reduces the computation time required to grow the tree, it cannot be applied to any predictor builder, oppositely to the bagging procedure.

When using a fixed number of processors, the training time and the prediction time grow linearly with respect to the number of predictors trained. However, this is not a major issue since experimental results exhibit small improvements in prediction when the number of trees grows above 50, see [Geurts *and al.* 2006].

The effect of using multiple trees is shown in Figure 2.2. In this figure, 25 inputs are sampled at uniform random in $[-\pi, \pi]$, and $y = \sin(x)$. Three different regression forests are built using respectively 1, 10 and 100 regression trees. The observations used to train the models are displayed as black circles, the prediction according to the regression forests are shown with different types of lines and the real function from which observations are drawn is shown as a thick transparent line. The prediction is much smoother when using multiple trees and the difference with the sampled function tends to be lower.

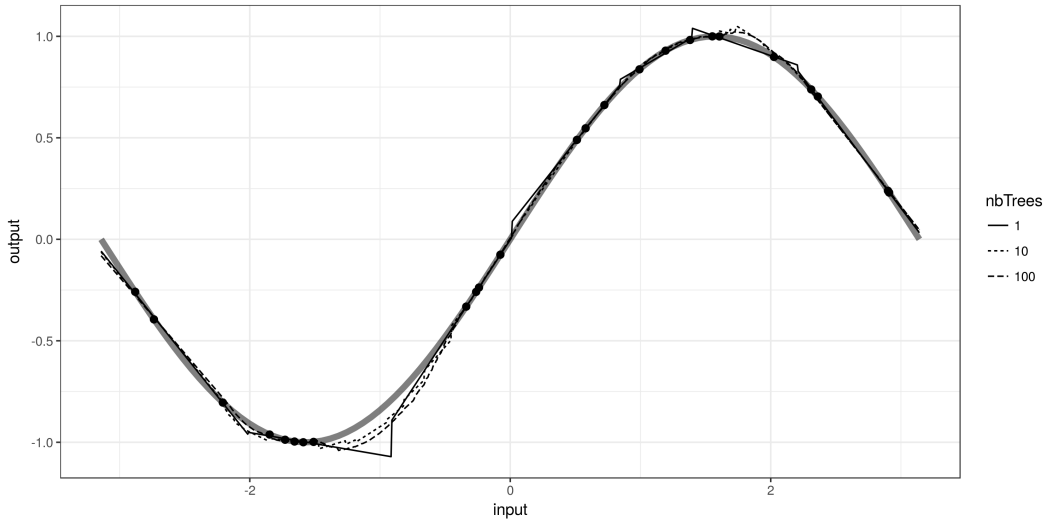


Figure 2.2 – An example of prediction for regression forests with linear models in leaves

An in-depth survey on classification and regression forests is presented in [Loh 2011].

2.1.5 ExtraTrees

The ExtraTrees algorithm has been proposed in [Geurts *and al.* 2006], it grows a regression forest from samples. In this section, we start by defining notions re-

quired to understand this algorithm and introducing modifications we brought to the original algorithm. Finally, we present the ExtraTrees algorithm used in this thesis.

We denote a training set $\mathcal{TS} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ with $x_k \in \mathbb{R}^n$ and $y_k \in \mathbb{R}$.

The function splitTrainingSet separates a training set $\mathcal{TS}$ using an univariate split (d, v). It is described in Algorithm 4.

Algorithm 4 Separating a training set

function splitTrainingSet($\mathcal{TS}$,d,v)
    ▷ Parameters description:
        $\mathcal{TS}$ The training set to be split
        d The input dimension of the split
        v The value of the split
    $\mathcal{TS}_L \leftarrow \{\}$
    $\mathcal{TS}_U \leftarrow \{\}$
    for all $(x, y) \in \mathcal{TS}$ **do**
        if $x_d \leq v$ **then**
            $\mathcal{TS}_L$.insert((x, y))
        else
            $\mathcal{TS}_U$.insert((x, y))
        end if
    end for
    return $(\mathcal{TS}_L, \mathcal{TS}_U)$ ▷ A partition of $\mathcal{TS}$ according to split (d, v)
end function

Consider a training set $\mathcal{TS}$, we denote outputMean($\mathcal{TS}$) the average of the outputs of the training set, see Eq. (2.3), and outputVar($\mathcal{TS}$) the variance of the outputs of the training set, see Eq. (2.4).

$$\text{outputMean}(\mathcal{TS}) = \frac{\sum_{(x,y)\in\mathcal{TS}} y}{|\mathcal{TS}|} \tag{2.3}$$

$$\text{outputVar}(\mathcal{TS}) = \frac{\sum_{(x,y)\in\mathcal{TS}} (y - \text{outputMean}(\mathcal{TS}))^2}{|\mathcal{TS}|} \tag{2.4}$$

While the original ExtraTrees algorithm uses piecewise constant approximation, PWC for short, we allow both piecewise constant and piecewise linear approximations, PWL for short. Consider $\mathcal{TS}$ a training set and type $\in \{\text{PWC}, \text{PWL}\}$, we obtain the approximator through the use of the fitModel

function:

$$\textsc{fitModel}(\mathcal{TS}, \text{type}) = \begin{cases} \mathcal{C}\left(\textsc{outputMean}(\mathcal{TS})\right) & \text{if type } = \text{PWC} \\ \mathcal{L}\left(A, B\right) & \text{if type } = \text{PWL} \end{cases}$$

where A and B are obtained through least square regression for the PWL case.

In order to favor trees which strongly reduces the variance, ExtraTrees computes the score of a split according to Algorithm 5, which is denoted evalSplitScore. Lower scores are considered as better.

The core of the ExtraTrees algorithm is based on its procedure for growing trees. We denote this procedure growExtraTree and describe it in Algorithm 6. It consists of repeatedly separating the training set in two by the best random split chosen among the k available, with k a parameter of the algorithm.

In the original version of ExtraTrees, $n_{\min}$ represents the required number of samples to split a given node. Since we allow the use of linear models in leaves, it is not sufficient. A strong guarantee on the number of samples in a single node is required. Therefore, we brought the following modification. When sampling the split position along a dimension d, we restrict the possible set for split values in $[s_{\min}, s_{\max}]$, where $s_{\min}$, resp. $s_{\max}$ is the $n_{\min}$-th smallest, resp. highest, value of the inputs of the training set along the dimension d. Finally, if $s_{\min} < s_{\max}$, we choose the split dimension at uniform random in $[s_{\min}, s_{\max}[$, otherwise we forbid the split.

Due to the terminal conditions of the original version of ExtraTrees, large trees were grown for parts where the output is almost constant, while a single leaf would already lead to similar performances. While growing large trees does not impact the quality of the approximation, it impacts the number of nodes and therefore the access time, two crucial aspects for real-time applications. In order to reduce the size of the trees, we introduce an additional parameter: $v_{\min}$. It is used for an additional terminal condition: if the variance of outputs in a training set is smaller or equal to $v_{\min}$, then the algorithm stops dividing the set.

Since growExtraTree is a stochastic algorithm, it allows to grow several trees and then average them for a more accurate prediction. The ExtraTrees algorithm, see Algorithm 7, grows m trees independently, where m is a parameter of the algorithm. An in-depth study of the impact of this parameter is presented in [Geurts *and al.* 2006]. It suggests that the gain of using more than 50 trees inside the forest is very low.

Since adjusting the parameters of a regression algorithm is a time-consuming task, we propose the following scheme for the parameters: k is equal to the input space dimension, $n_{\min}$ grows logarithmically with respect to the size of the training set and $v_{\min}$ is 0 since it is difficult to anticipate the relevant scale for variance.

Algorithm 5 The evaluation of splits for ExtraTrees

function evalSplitScore($\mathcal{TS}$,d,v,type)
▷ Parameters description:

$\mathcal{TS}$	The set of samples
d	The input dimension of the split
v	The value of the split
type	The type of approximation used

$(\mathcal{TS}_L, \mathcal{TS}_U) \leftarrow$ splitTrainingSet$(\mathcal{TS}, d, v)$
$\tilde{f}_L \leftarrow$ fitModel$(\mathcal{TS}_L, \text{type})$
$\tilde{f}_U \leftarrow$ fitModel$(\mathcal{TS}_U, \text{type})$
$e_L \leftarrow \sum_{(x,y)\in\mathcal{TS}_L}(y - \tilde{f}_L(x))^2$
$e_U \leftarrow \sum_{(x,y)\in\mathcal{TS}_U}(y - \tilde{f}_U(x))^2$
return $e_L + e_U$ ▷ The score of split (d, v)
end function

While the algorithm presented here allows to build a function approximator from $\mathbb{R}^n$ to $\mathbb{R}$, note that it is possible to approximate a function with a multi-dimensional output space by considering that all the output dimensions are independent and growing a regression forest for each output dimension.

2.1.6 Projection of a function approximator

Consider a function approximator $\tilde{f}$ with input space $I \times J$ and output space O. Typically $\tilde{f}$ may be a policy, I the state space and J the action space. Then for every $i \in I$, f can be projected as $\mathcal{P}\left(\tilde{f}, i\right) : J \to O$ defined by:

$$\mathcal{P}\left(\tilde{f}, i\right)(j) = \tilde{f}(i, j) \ .$$

In case the approximator is represented by a regression tree, its projection is a smaller regression tree, computable in linear time as follows.

Consider a constant model $\tilde{f} = \mathcal{C}(o)$ with $o \in O$, since its value is already constant, projecting does not affect the prediction, therefore $\mathcal{P}(\mathcal{C}(o), i) = \mathcal{C}(o)$

Consider a linear model $l = \mathcal{L}(A, B)$, with input space $I \times J$ and output space O. We can rewrite $A = \begin{bmatrix} A_i & A_j \end{bmatrix}$ with A_i the first $\dim_R(I)$ columns of A and A_j the last $\dim_R(J)$ columns of A. Then $\mathcal{P}(\mathcal{L}(A, B), i) = \mathcal{L}(A_j, B + A_i i)$.

Consider a regression tree $t = \mathcal{T}\left(d, v, \tilde{f}_1, \tilde{f}_2\right)$ with input space $I \times J$ and output space $\mathbb{R}^n$. Projection of the tree t on $i \in I$ is performed according to algorithm 8, and is denoted projectTree(t, i).

thus producing a regression tree of size smaller than $\mathcal{T}\left(d, v, \tilde{f}_1, \tilde{f}_2\right)$.

Algorithm 6 The GROWEXTRATREE algorithm

function GROWEXTRATREE($\mathcal{TS}$,type,k,$n_{\min}$,$v_{\min}$)
  ▷ Parameters description:
    $\mathcal{TS}$ The set of samples
    type The type of leaves to be used
    k The number of dimensions to try
    $n_{\min}$ The minimal number of samples per node
    $v_{\min}$ The minimal variance inside a node to allow split
  if $|\mathcal{TS}| < 2n_{\min} \vee$ OUTPUTVAR($\mathcal{TS}$) $< v_{\min}$ **then**
    return FITMODEL($\mathcal{TS}$, type)
  end if
  dimCandidates $\leftarrow$ sample k distinct elements in $\{1, \ldots, \dim_R(\mathcal{H})\}$
  bestSplit $\leftarrow$ NULL
  bestScore $\leftarrow \infty$
  for all $d \in$ dimCandidates **do**
    $s_{\min} \leftarrow$ the $n_{\min}$ smallest value of X along dimension d
    $s_{\max} \leftarrow$ the $n_{\min}$ highest value of X along dimension d
    if $s_{\min} \geq s_{\max}$ **then**
      continue
    end if
    $v \leftarrow$ SAMPLE($\mathcal{U}([s_{\min}, s_{\max}])$)
    score $\leftarrow$ EVALSPLITSCORE($\mathcal{TS}, d, v$, type)
    if score $<$ bestScore **then**
      bestSplit $\leftarrow (d, v)$
      bestScore $\leftarrow$ score
    end if
  end for
  if bestSplit = NULL **then** ▷ Might happen if $s_{\min} = s_{\max}$ for all d
    return FITMODEL(X, Y, type)
  end if
  $(d, v) \leftarrow$ bestSplit
  $(\mathcal{TS}_L, \mathcal{TS}_U) \leftarrow$ SPLITTRAININGSET($\mathcal{TS}, d, v$)
  $t_L \leftarrow$ GROWEXTRATREE($\mathcal{TS}_L$, type, $n_{\min}, v_{\min}$)
  $t_U \leftarrow$ GROWEXTRATREE($\mathcal{TS}_U$, type, $n_{\min}, v_{\min}$)
  return $\mathcal{T}(d, v, t_L, t_U)$ ▷ A regression tree based on $\mathcal{TS}$
end function

Consider a regression forest $f = \mathcal{F}(t_1, \ldots, t_n)$ with input space $I \times J$ and output space $\mathbb{R}^n$. Projection of the forest f on $i \in I$ is performed as following:

$$\mathcal{P}(\mathcal{F}(t_1, \ldots, t_n), i) = \mathcal{F}(\mathcal{P}(t_1, i), \ldots, \mathcal{P}(t_n, i))$$

Algorithm 7 The ExtraTrees algorithm

function ExtraTrees($\mathcal{TS}$,type,k,$n_{\min}$,$v_{\min}$,m)
▷ Parameters description:

$\mathcal{TS}$	The set of samples
type	The type of leaves to be used
k	The number of dimensions to try
$n_{\mathbf{min}}$	The minimal number of samples per node
$v_{\mathbf{min}}$	The minimal variance inside a node to allow split
m	The number of trees

    trees$\leftarrow \{\}$
    for all $i \in \{1, \ldots, m\}$ **do**
        trees.insert(growExtraTree($\mathcal{TS}$, type, $n_{\min}$, $v_{\min}$)
    end for
    return $\mathcal{F}$(trees) ▷ A regression forest based on $\mathcal{TS}$
end function

Algorithm 8 The projectTree algorithm

function projectTree(t,i)
▷ Parameters description:

t	The regression tree to project
i	The value on which the tree is projected

    if isLeaf(t) **then**
        return $\mathcal{P}(t, i)$ ▷ t is a constant or linear approximator
    end if
    $d \leftarrow$ splitDim(t)
    $v \leftarrow$ splitVal(t)
    $D \leftarrow \dim_R(i)$
    if $d > D$ **then**
        $t_L \leftarrow$ projectTree(LC(t), i)
        $t_U \leftarrow$ projectTree(UC(t), i)
        return $\mathcal{T}(d - D, v, t_L, t_U)$
    else if $i_d \leq v$ **then**
        return projectTree(LC(t), i)
    else
        return projectTree(UC(t), i)
    end if
end function

2.1.7 Weighted average of function approximators

Consider two function approximators $\tilde{f}_1$,$\tilde{f}_2$ with the same input space $X \in \mathbb{R}^n$ and the same output space. Consider two weights $w_1 \in \mathbb{R}^+$ and $w_2 \in \mathbb{R}^+$. We denote $\tilde{f}' = \text{wAvg}\left(\tilde{f}_1, \tilde{f}_2, w_1, w_2\right)$ the weighted average of $\tilde{f}_1$ and $\tilde{f}_2$ by w_1 and w_2. It follows equation (2.5).

$$\forall x \in X, \tilde{f}'(x) = \frac{\tilde{f}_1(x)w_1 + \tilde{f}_2(x)w_2}{w_1 + w_2} \tag{2.5}$$

Computing a symbolic expression for weighted average of constant and linear model is trivial. We present the weighted average of two constant models in equation (2.6), the weighted average of two linear models in equation (2.7) and the weighted average of one constant and one linear model in equation (2.8).

$$\text{wAvg}\left(\mathcal{C}\left(o_1\right), \mathcal{C}\left(o_2\right), w_1, w_2\right) = \mathcal{C}\left(\frac{o_1 w_1 + o_2 w_2}{w_1 + w_2}\right) \tag{2.6}$$

$$\text{wAvg}\left(\mathcal{L}\left(A_1, B_1\right), \mathcal{L}\left(A_2, B_2\right), w_1, w_2\right) = \mathcal{L}\left(\frac{A_1 w_1 + A_2 w_2}{w_1 + w_2}, \frac{B_1 w_1 + B_2 w_2}{w_1 + w_2}\right) \tag{2.7}$$

$$\text{wAvg}\left(\mathcal{L}\left(A, B\right), \mathcal{C}\left(o\right), w_1, w_2\right) = \mathcal{L}\left(A w_1, \frac{B w_1 + o w_2}{w_1 + w_2}\right) \tag{2.8}$$

Computing the weighted average of two trees t_1 and t_2 is more complex. A simple scheme for computing $t' = \text{wAvg}\left(t_1, t_2, w_1, w_2\right)$ would be to root a replicate of t_2 at each leaf of t_1 and to average the final approximators. However this would lead to an overgrown tree containing unreachable nodes. As example, a split with the predicate $x_1 \leq 3$ could perfectly appear on the lower child of another node whose predicate is $x_1 \leq 2$.

We designed an algorithm called WEIGHTEDAVERAGE for computing the weighted average of two functions approximators represented by regression trees. This is Algorithm 9. The algorithm walks simultaneously both trees from the roots to the leaves, and performs on-the-fly optimizations. This algorithm runs in linear time in the worst case and in logarithmic time on well-balanced trees. It tends to preserve the original aspect of the regression tree with top-most nodes carrying the most important splits (i.e. splits that strongly reduce the variance of their inner sets of samples). A graphical example of input and output of the algorithm is shown in Figure 2.3.

2.1.8 Pruning trees

Pruning is a procedure which aims at reducing the size of a regression tree while minimizing the loss of precision of the corresponding function approximator.

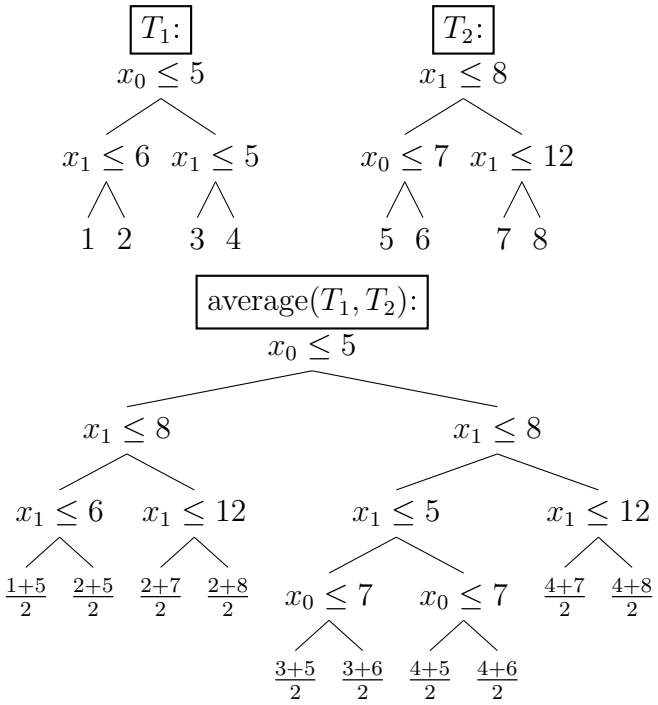


Figure 2.3 – An example of tree merge

We define the notion of loss received by replacing an approximator $\tilde{f}$ by an approximator $\tilde{f}'$ on space X as:

$$\textsc{Loss}(\tilde{f}, \tilde{f}', X) = \int_{x \in \mathcal{H}} \|\tilde{f}'(x) - \tilde{f}(x)\| \mathrm{d}x \tag{2.9}$$

We designed an algorithm called PRUNETREE, which takes as input a parameter n and a regression tree t and prunes the tree t to produce a regression tree with at most n leaves. This is Algorithm 10.

The algorithm PRUNETREE considers only *preleaves* as candidates for pruning. A preleaf is an inner node of the tree (i.e. not a leaf) whose both left and right sons are leaves. Remark that since all inner nodes of a regression trees have exactly two sons, then every regression tree has at least one preleaf. The function GETPRELEAVES retrieves all the preleaves from a tree:

$$\begin{aligned}&\textsc{getPreLeaves}(t) = \\ &\qquad \{(t' \text{ node of } t \ \mid \neg\textsc{isLeaf}(t') \wedge \textsc{isLeaf}(\mathrm{UC}(t')) \wedge \textsc{isLeaf}(\mathrm{LC}(t'))\} \ .\end{aligned}$$

Algorithm PRUNETREE, see Algorithm 11, prunes a preleaf by computing the average of its lower and upper sons, weighted by the volume of the corresponding hyperrectangles. Thus the algorithm relies on a constant time primitive $\mathrm{wAvg}\,(\mathrm{LC}, \mathrm{UC}, w_L, w_U)$ computing this weighted average, when LC and UC are constant or linear models.

Algorithm 9 Computing the weighted average of two regression trees

```
1: function WEIGHTEDAVERAGE(t_1, t_2, w_1, w_2)
2:         ▷ Parameters description:
               t_1,t_2   The regression trees
               w_1,w_2   The weights
3:     if ISLEAF(t_1) then
4:         if ISLEAF(t_2) then
5:             t' ← wAvg(t_1, t_2, w_1, w_2)
6:         else
7:             t' ← WEIGHTEDAVERAGE(t_2, t_1, w_2, w_1)
8:         end if
9:     else                                              ▷ t_1 is not a leaf
10:        v_m ← min along dim d from H(t_1)
11:        v_M ← max along dim d from H(t_1)
12:        d ← SPLITDIM(t_1)
13:        v ← SPLITVAL(t_1)
14:        if v_M ≤ v then                               ▷ Upper child is useless
15:            t' ← WEIGHTEDAVERAGE(t_2, LC(t_1), w_2, w_1)
16:        else if v_m < v then                          ▷ Lower child is useless
17:            t' ← WEIGHTEDAVERAGE(t_2, UC(t_1), w_2, w_1)
18:        else
19:            t_L ← WEIGHTEDAVERAGE(t_2, LC(t_1), w_2, w_1)
20:            t_U ← WEIGHTEDAVERAGE(t_2, UC(t_1), w_2, w_1)
21:            t' ← T(d, v, t'_L, t'_U)
22:        end if
23:    end if
24:    return t'
25: end function
```

The algorithm maintains a list of potential candidates for pruning, ordered by loss, in order to get quick access to the candidate which brings the lowest loss. For that it relies on the function GETPRUNINGCANDIDATES defined as Algorithm 10. GETPRUNINGCANDIDATES performs a greedy selection of candidates for pruning, starting with those with the lowest loss. This algorithm ensures a reasonable computation time, even though it may not produce an optimal pruning.

While most pruning procedures in the literature aim at reducing the risk of overfitting with respect to a given learning and testing sets, PRUNETREE is independent of these sets. Our algorithm focuses on time-efficiency and limiting the size of the output tree, with no guarantee against overfitting.

Algorithm 10 Gathering candidates for tree pruning

function GETPRUNINGCANDIDATES(t)
    ▷ Parameters description:
        t The tree in which pruning candidates should be searched for
    candidates $\leftarrow$ () ▷ Ordered set of tuples $(t, l, \tilde{f})$, with t a regression tree, $l \in \mathbb{R}$ the loss, and $\tilde{f}$ a function approximator. The set is ordered by loss.
    for all $t' \in$ GETPRELEAVES(t) **do**
        $w_L \leftarrow |\mathcal{H}(\mathrm{LC}(t'))|$
        $w_U \leftarrow |\mathcal{H}(\mathrm{UC}(t'))|$
        $\tilde{f}' \leftarrow \mathrm{wAvg}\left(\mathrm{LC}(t), \mathrm{UC}(t), w_L, w_U\right)$
        $l \leftarrow$ LOSS$(t', f', \mathcal{H}(t'))$ ▷ see Eq. (2.9)
        add $(t', l, \tilde{f}')$ to candidates
    end for
    return candidates
end function

Algorithm 11 The tree pruning algorithm

function PRUNETREE(t,$n_{\max}$)
    ▷ Parameters description:
        t The tree which need to be pruned
        $n_{\max}$ The maximal number of leaves allowed
    nbLeaves $\leftarrow$ NBLEAVES(t)
    candidates $\leftarrow$ GETPRUNINGCANDIDATES(t) ▷ Ordered, see Alg. 10
    while nbLeaves $> n_{\max}$ **do**
        $(t', l, \tilde{f}') \leftarrow$ pop(candidates) ▷ Pop lowest loss candidate
        $t' \leftarrow \tilde{f}'$ ▷ Replace the preleaf by the approximation model
        if t' has an ancestor **then**
            newCandidates $\leftarrow$ GETPRUNINGCANDIDATES(ANCESTOR(t'))
            candidates $\leftarrow$ candidates $\cup$ newCandidates
        end if
        nbLeaves $\leftarrow$ nbLeaves $- 1$
    end while
    return t ▷ The pruned tree
end function

2.1.9 From a forest back to a tree

Any regression forest can be converted back to a regression tree by using successive merges with the function WEIGHTEDAVERAGE, see Algorithm 9. However the problem is that the number of leaves in the resulting tree can

be exponential in the size of the forest: it may grow up to the product of the number of leaves of all trees of the forest.

In order to reduce the size of the tree produced, we accept a loss of precision and use the PRUNETREE function (see Algorithm 11) to reduce the size of the tree produced. By iterating the processes of pruning and merging, MERGEFORESTWITHPRUNING allows to extract a single tree from a forest. This procedure is described in Algorithm 12.

The procedure MERGEFORESTWITHPRUNING ensures that when merging two trees, each tree has at most n leaves, with n a parameter of the procedure. Thus, at any time during the procedure, the maximal number of leaves in a tree is n^2. Therefore, the complexity of the procedure is in $O(n^2 m)$, with $m = \text{NBTREES}(f)$.

Algorithm 12 Merging a forest in a single tree

```
1: function MERGEFORESTWITHPRUNING(f,n)
2:             ▷ Parameters description:
                  f   The forest which will be merged
                  n   The number of leaves allowed for the final tree
3:     t ← PRUNETREE(GETTREE(f, 1), n)
4:     for all i ∈ {2, ..., NBTREES(f)} do
5:         t' ← PRUNETREE(GETTREE(f, i), n)
6:         t ← WEIGHTEDAVERAGE(t, t', i − 1, 1)
7:         t ← PRUNETREE(t, n)
8:     end for
9:     return t
10: end function
```

2.2 Fitted Policy Forest

When there is no black-box model available for a learning problem, it is not possible to solve it offline. Thus, it is mandatory to obtain samples by interacting with the system.

We present here the Fitted Policy Forest algorithm, which can perform batch-mode learning or semi-online learning. This algorithm is mainly based on the use of regression forests to approximate both, the Q-value and the policy. While it can handle continuous action spaces, it cannot deal with heterogeneous action spaces.

We start by describing the algorithm for batch-mode learning, then we show how we can use a simple exploration scheme to transform it into a semi-online learning algorithm. Experimental results on classical learning problems

are presented for both batch mode learning and semi-online learning. Finally, we discuss the applicability of FPF to our targeted problems.

2.2.1 Batch-mode algorithm

The batch learning algorithm FPF, is mainly based on the Fitted Q Iteration algorithm, see [Ernst *and al.* 2005]. Fitted Q Iteration, FQI for short, is entirely based on the access to a batch of data $\mathcal{D} = \{(s_1, a_1, s'_1, r_1), \ldots, (s_n, a_n, s'_n, r_n)\}$, with $s_k \in S$ the initial state of sample k, $a_k \in A$ the action of sample k, $s'_k \in S$ the state measured after applying a_k from state s_k and $r_k \in \mathbb{R}$ the reward received when reaching state s'_k. These data are used iteratively by a regression algorithm to produce an approximation of the Q-value function according to Algorithm 13.

Algorithm 13 Fitted Q Iteration algorithm

1: **function** FITTEDQITERATION($\mathcal{D}$,N)
2: ▷ Parameters description:
$\mathcal{D}$ The set of samples used for batch learning
N The number of iterations of the algorithm
3: $\tilde{Q}_0 \leftarrow \mathcal{C}(0)$
4: $i \leftarrow 0$
5: **while** $i < N$ **do**
6: $i \leftarrow i + 1$
7: $X \leftarrow \left\{ \begin{bmatrix} s_k \\ a_k \end{bmatrix} \mid k \in \{1, \ldots, |\mathcal{D}|\} \right\}$
8: $Y \leftarrow \left\{ r_k + \gamma \max_{a \in A} \tilde{Q}_{n-1}(s'_k, a) \mid k \in \{1, \ldots, |\mathcal{D}|\} \right\}$
9: $\tilde{Q}_n \leftarrow$ EXTRATREES($\mathcal{TS}(X, Y)$, PWC) ▷ default configuration
10: **end while**
11: **return** $\tilde{Q}_n$
12: **end function**

At line 9 of Algorithm 13, EXTRATREES is used to approximate the Q-value function. We suggest to use constant approximators in leaves, because both experimental results and literature suggest that using linear approximators might lead to a divergent Q-value estimation, see [Ernst *and al.* 2005]. Note that it is perfectly possible to use any other regression methods.

While this procedure yields very satisfying results when the action space is discrete, the computational complexity of the max part at line 8 of Algorithm 13 is a problem when using regression forests. Therefore, in [Ernst *and al.* 2005], action spaces are always discretized to compute this equation,

thus leading to an inappropriate action set when optimal control requires a fine discretization or when the action space is multi-dimensional.

In order to estimate the best action a from state s using the approximation $\tilde{Q_{n-1}}$, we use Algorithm 14, named ACTIONFROMQVALUE. First of all, since we know s, we can use projection on function approximators to reduce the size of the forest Q_{n-1}. Then, we can simply merge all the trees of the forest f into a single tree t, at the cost of precision. Once a tree is obtained, we can iterate on all its leaves to find out the maximum, since finding the maximum of a linear model inside an hyperrectangle is trivial. Note that if the leaf l containing the maximal value of the tree is a constant approximator, we consider that arg max returns CENTER($\mathcal{H}(l)$). While this procedure gives only an approximation of the best action based on the Q-value it is already more accurate and scalable than using a simple discretization.

Algorithm 14 Choosing an action from a Q-value approximation

1: **function** ACTIONFROMQVALUE($\tilde{Q}$,s,n,A)
2: ▷ Parameters description:
$\tilde{Q}$ A regression forest representing the Q-value
s The current state
$n_{\max}$ The number of leaves allowed for the final tree
A The action space, an hyperrectangle
3: $Q' \leftarrow \mathcal{P}\left(\tilde{Q}, s\right)$ ▷ Q' is a regression forest from A to $\mathbb{R}$
4: $t \leftarrow$ MERGEFORESTWITHPRUNING($Q', n_{\max}$)
5: **return** $\arg\max_{a \in A} t(a)$ ▷ Tree-search of highest value
6: **end function**

Once the estimation of the Q-value $\tilde{Q}$ is obtained, FQI proposes to derive the policy based on Eq. 2.10.

$$\tilde{\pi^*}(s) = \arg\max_{a \in A} \tilde{Q}(s, a) \tag{2.10}$$

We propose to use ACTIONFROMQVALUE to estimate the best action, see Algorithm 14. However, since this method require a projection of the forest and then a merge with pruning, it is computationally expensive. Since online cost of accessing the policy is crucial in robotics, we developed a new method named *fitted policy forests*, FPF for short.

Once the Q-value is approximated by a forest, FPF samples states $s \in S$ randomly, find an estimate of the best action $a \in A$ and then use the collected couples (s, a) to grow a forest which will represent a policy. We propose two different versions of FPF. First, *FPF:PWC* in which the models in the leaves

are constant and second *FPF:PWL* in which the models in the leaves are linear. The FPF algorithm is introduced in Algorithm 15 and a flowchart representing the differences between FQI and FPF is shown in Fig. 2.4.

Algorithm 15 The fitted policy forests algorithm

function FPF($\mathcal{D}$,n,N,M, S,A,type)
    ▷ Parameters description:
        $\mathcal{D}$ The set of samples used for batch learning
        n The number of leaves authorized for merging procedures
        N The number of iterations of the FQI algorithm
        M The number of states used to approximate the policy
        S The state space of the MDP
        A The action space of the MDP
        type The type of approximator to use
    $\tilde{Q} \leftarrow$ FITTEDQITERATIONALGORITHM($\mathcal{D}, N$)
    $\mathcal{TS} \leftarrow \{\}$
    while $|\mathcal{TS}| < M$ **do**
        $s \leftarrow$ sample S uniformly
        $a \leftarrow$ ACTIONFROMQVALUE($\tilde{Q}, s, n, A$)
        add (s, a) to $\mathcal{TS}$
    end while
    return EXTRATREES($\mathcal{TS}$, type) ▷ The policy computed by FPF
end function

Since FQI requires to project trees and merge forests, it is computationally too expensive to be used in online situations. On the other hand, both FPF methods only require to find the appropriate leaf of each tree. Its computational complexity grows linearly with respect to the number of trees and logarithmically with respect to the number of leaves per tree, thus making it entirely suited for online use in robotics application.

2.2.2 Batch-mode experiments

We used three benchmark problems classical in reinforcement learning to evaluate the performances of the FPF algorithms. While all the methods share the same parameters for computing the Q-value forest, we tuned specifically parameters concerning the approximation of the policy using the Q-value forest. We compared our results with those presented in [Pazis and Lagoudakis 2009], however we do not have access to their numerical data, and rely only on the graphical representation of these data. Thus, the graphical lines shown for

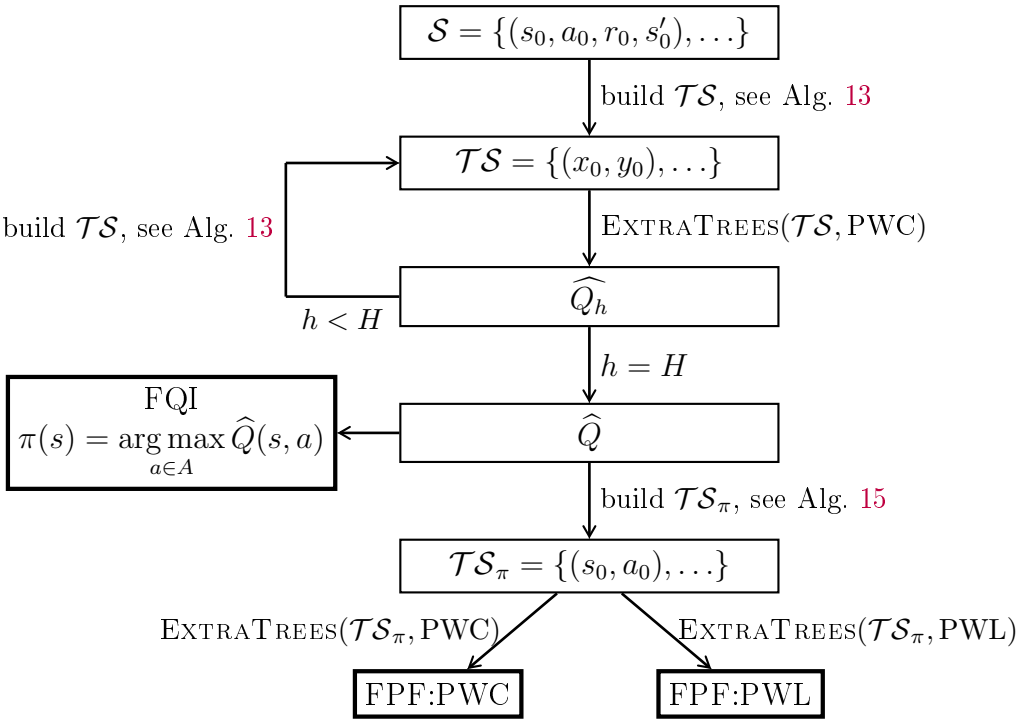


Figure 2.4 – A flowchart of the different methods

BAS are approximate and drawn thicker than the other to highlight the noise in measurement. We present the result separately for the three benchmarks while discussing results specific to a problem as well as global results.

On all the problems, performances of FPF:PWL are better or at least equivalent to those achieved by BAS in [Pazis and Lagoudakis 2009].

This is remarkable, because BAS uses a set of basic functions specifically chosen for each problem, while our method is generic for all the problems. The computation cost of retrieving actions once the policy has been calculated appears as negligible and therefore confirms that our approach is well-suited for high-frequency control in real-time.

Inverted pendulum stabilization

The *inverted pendulum stabilization* problem consists of balancing a pendulum of unknown length and mass by applying a force on the cart it is attached to. An uniform noise is added at every time step and the cost function includes: the position of the pendulum, the angular velocity of the pendulum and the force applied on the cart. Therefore, policies resulting with smoother motions perform better than bang-bang policies. A complete description of the problem is provided in Section 1.1.3.

The training sets were obtained by simulating episodes using a random policy, and the maximal number of steps for an episode was set to 3000. The

performances of the policies were evaluated by testing them on episodes of a maximal length of 3000 and then computing the cumulative reward. In order to provide an accurate estimate of the performance of the algorithms, we computed 50 different policies for each point displayed in Figure 2.5 and average their cumulative reward (vertical bars denote 95% confidence interval). The parameters used to produce the policies are shown in Table 2.1.

FPF clearly outperforms FQI on this problem and PWL approximations outperform PWC approximations. Results for BAS [Pazis and Lagoudakis 2009] rank systematically lower than both FPF methods. The huge difference of learning speed between FQI and FPF suggests that using regression forests to learn the policy from the Q-value can lead to drastic improvements. On such a problem where the optimal policy requires a fine choice of action, it is not surprising that using linear models to represent the policy provides higher results than constant models.

The best value for $n_{\min}$, the minimal number of samples per leaf, is pretty high (17 for PWC and 125 for PWL). Our understanding of this phenomena is that the Q-value tree tends to slightly overfit the data, additionally, it uses PWC approximation. Therefore, using it directly leads to an important quantification noise. Using a large value for $n_{\min}$ might be seen as applying a smoothing, which is, according to [Ernst *and al.* 2005], considered as necessary for regression trees sampling a stochastic function. The need for a large number of samples is increased for FPF:PWL, because providing a reliable linear interpolation of a noisy application requires far more samples than a constant interpolation.

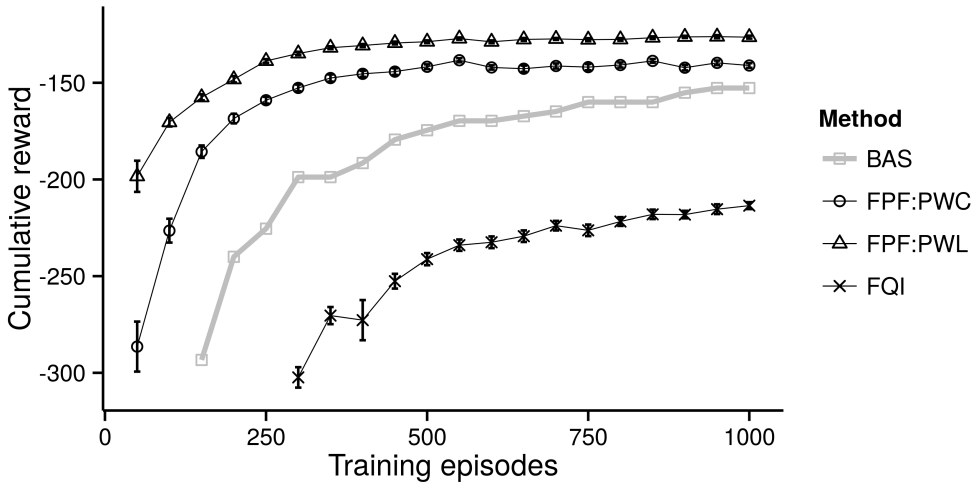


Figure 2.5 – Performance on the Inverted Pendulum Stabilization problem

Table 2.1 – Parameters used for Inverted Pendulum Stabilization

Parameter	FQI	FPF:PWC	FPF:PWL
Nb Samples	NA	10'000	10'000
Max Leaves	50	50	50
k	NA	2	2
$n_{\min}$	NA	17	125
M	NA	25	25
$V_{\min}$	NA	10^{-4}	10^{-4}

Double integrator

In order to provide a meaningful comparison, we stick to the description of the problem given in [Pazis and Lagoudakis 2009] where the control step has been increased from the original version presented in [Santamaria *and al.* 1997]. The double integrator is an episodic problem in a single dimension, the agent has to reach a given position while minimizing a cost function based on the distance to the target and the acceleration used. A complete definition of the problem can be found in Section 1.1.1.

The training sets were obtained by simulating episodes using a random policy, and the maximal number of steps for an episode was set to 200. The performances of the policies were evaluated by testing them on episodes of a maximal length of 200 and then computing the cumulative reward. In order to provide an accurate estimate of the performance of the algorithms, we computed 100 different policies for each point displayed in Figure 2.6 and average their results. The parameters used for learning the policy are shown in Table 2.2.

On this problem, although none of the proposed methods reach BAS performance when there are more than 300 learning episodes, FPF:PWL learns quicker than BAS with a small number of episodes. It is important to note that while our basic function approximator is constant, the implementation of BAS used in [Pazis and Lagoudakis 2009] relies on expert parametrization of the set of policies, more precisely on the fact that the optimal policy is known to be a linear-quadratic regulator [Santamaria *and al.* 1997].

Car on the hill

While there has been several definitions of the *Car on the Hill* problem, we will stick to the version proposed in [Ernst *and al.* 2005] which was also used as a benchmark in [Pazis and Lagoudakis 2009]. In this episodic problem an underactuated car must reach the top of a hill. A detailed description of the problem can be found in Section 1.1.2.

As stated in [Pazis and Lagoudakis 2009], this problem is one of the worst

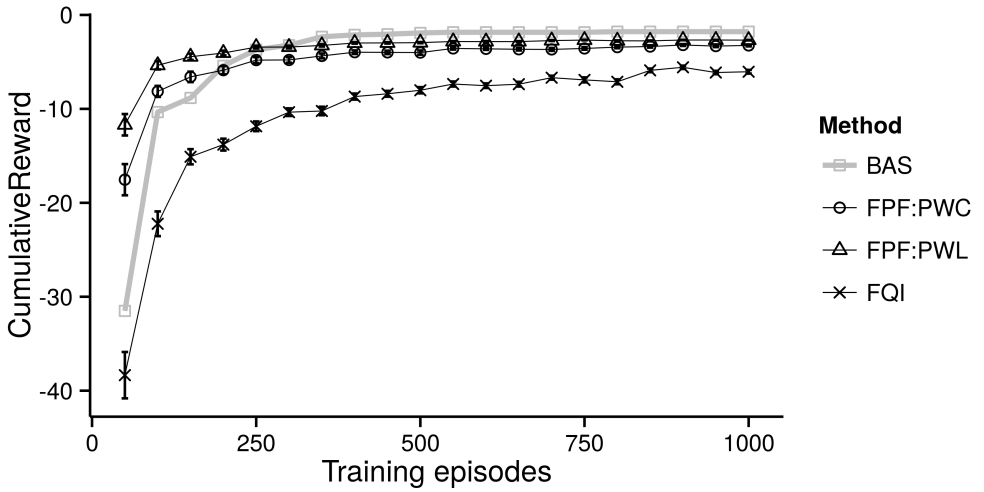


Figure 2.6 – Performance on the Double Integrator problem

Table 2.2 – Parameters used for Double Integrator

Parameter	FQI	FPF:PWC	FPF:PWL
Nb Samples	NA	10'000	10'000
Max Leaves	40	40	40
k	NA	2	2
$n_{\min}$	NA	100	1500
M	NA	25	25
$V_{\min}$	NA	10^{-4}	10^{-4}

cases for reinforcement learning with continuous action space. The reason is that the solution to this problem is a bang-bang strategy, i.e. a nearly optimal strategy exists which uses only two discrete actions: maximum acceleration and no acceleration. Thus the solver requires to learn a binary strategy composed of actions which have not been sampled frequently. It has been shown in [Ernst *and al.* 2005] that introducing more actions usually reduces the performances of the computed policies. Therefore, we do not hope to reach performances comparable to those achieved with a binary choice. This benchmark focuses on evaluating the performance of our algorithms, in one of the worst cases scenario.

While the samples of the two previous algorithms are based on episodes generated at a fixed initial state, the samples used for the *Car on the hill* problem are generated by sampling uniformly the state and action spaces. This procedure is the same which has been used in [Ernst *and al.* 2005] and [Pazis and Lagoudakis 2009], because it is highly improbable that a random policy could manage to get any positive reward in this problem. Evaluation is performed

by observing the repartition of the number of steps required to reach the top of the hill from the initial state.

Both our implementations of FPF and FQI strongly outperform BAS on the Car on the Hill problem. The histogram of the number of steps required for each method is represented on Figure 2.7. For each method, 200 different strategies were computed and tested. There is no significant difference in the number of steps required to reach the top of the hill between the different methods. For each method, at least 95% of the computed policies led to a number of step in the interval $[20, 25]$. Thus we can consider that an FPF or FQI controller takes 20 to 25 steps on average while it is mentioned in [Pazis and Lagoudakis 2009] that the controller synthesized by BAS requires 20 to 45 steps on average. Over the six hundred experiments gathered across three different methods, the maximal number of steps measured was 33.

FPF performs better than FQI on the problem Car on the Hill: although the number of steps required is not reduced by the FPF approach, the online cost is still reduced by around two orders of magnitude.

Computational cost

As mentioned previously, a quick access to the optimal action for a given state is crucial for real-time applications. We present the average time spent to retrieve actions for different methods in Figure 2.8 and the average time spent for learning the policies in 2.9. Experiments were run using an AMD Opteron(TM) Processor 6276 running at 2.3 GHz with 16 GB of RAM running on Debian 4.2.6. While the computer running the experiments had 64 processors, each experiment used only a single core.

We can see that using FPF reduces the average time by more than 2 orders of magnitude. Moreover, FPF:PWL presents a lower online cost than FPF:PWC, this is perfectly logical since representing a model using linear approximation instead of constant approximations requires far less nodes. While the results are only displayed for the "Double Integrator", similar results were observed for the two other problems.

It is important to note that the cost displayed in Figure 2.8 represents an entire episode simulation, thus it contains 200 action access and simulation steps. Therefore, it is safe to assume that the average time needed to retrieve an action with FPF:PWC or FPF:PWL is inferior to $50\mu s$. Even if the CPU used is two orders of magnitude slower than the one used in the experiment, it is still possible to include an action access at $200Hz$.

The additional offline cost of computing the polices required by FPF is lower than the cost of computing the Q-value using FQI when the number of training episode grows, as presented in Figure 2.9. Therefore, when it is possible to use FQI, it should also be possible to use FPF without significantly increasing the offline cost.

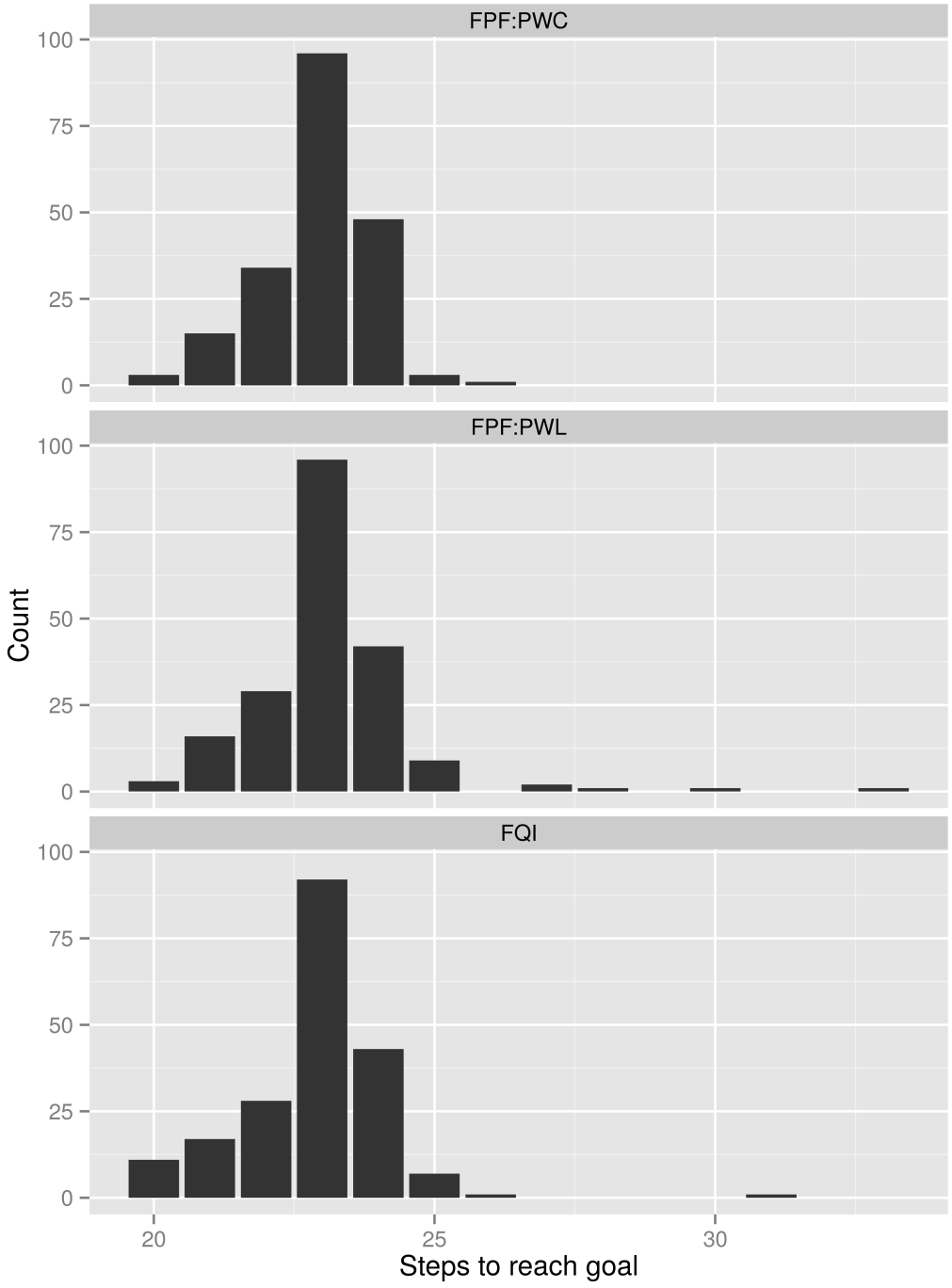


Figure 2.7 – Performance on the Car on the Hill problem

2.2.3 Semi-online algorithm

Since FPF is a batch algorithm, it can be used in semi-online learning mode simply by learning a new policy after every rollout with all the samples ac-

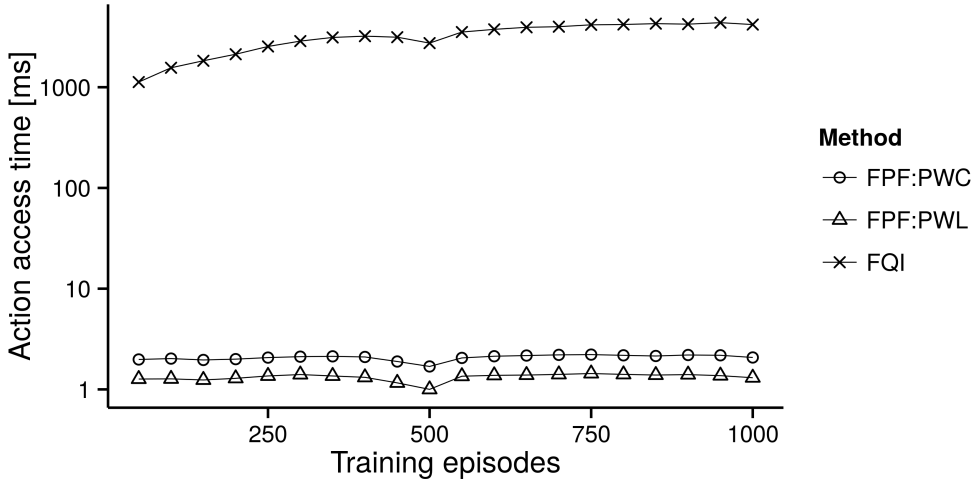


Figure 2.8 – Evaluation time by episode for the Double Integrator

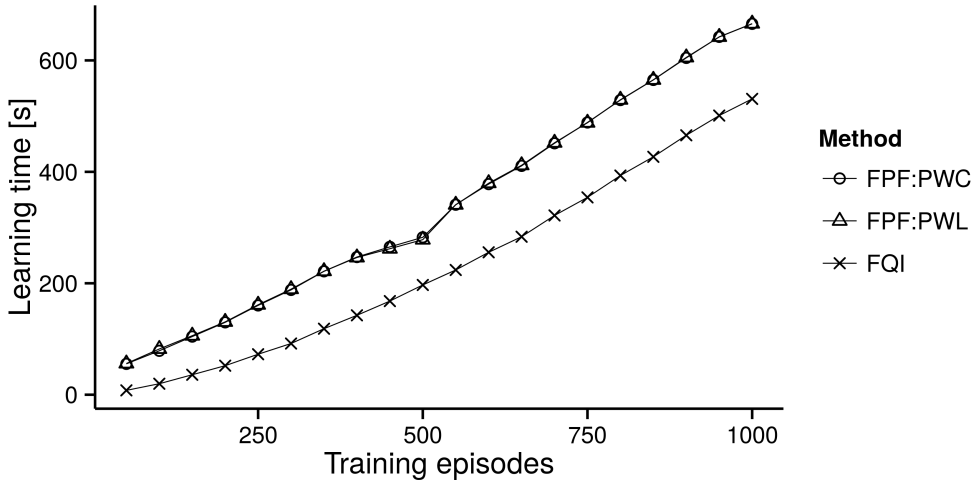


Figure 2.9 – Learning time by episode for the Double Integrator

quired. However, this procedure ignores the exploitation vs exploration trade-off by performing pure exploitation.

This section presents Multi-Resolution Exploration, MRE for short [Nouri and Littman 2009]. After introducing MRE, it presents several modifications we experimentally found to improve the performances of MRE when combined with FPF.

Original definition

MRE [Nouri and Littman 2009] proposes a generic algorithm allowing to balance the exploration and the exploitation of the samples. The main idea is to build a function $\kappa : S \times A \mapsto [0, 1]$ which estimates the knowledge amount gathered on a couple $(s, a) \in S \times A$. During the execution of the algorithm, when action a is taken in state s, a point $p = (s_1, \ldots, s_{\dim S}, a_1, \ldots, a_{\dim A})$ is inserted in a kd-tree, called knownness-tree. Then, the knownness value according to a knownness-tree τ at any point p can be computed by using Eq. (2.11).

$$\kappa(p) = \min\left(1, \frac{|P|}{\nu} \frac{1}{\left\lfloor \sqrt[k]{nk/\nu} \right\rfloor} \right) \tag{2.11}$$

where ν is the maximal number of points per leaf, $k = \dim(S \times A)$, n is the number of points inside the whole tree, $P = \text{points}(\text{leaf}(\tau, p,))$ and $\mathcal{H} = \text{space}(\text{leaf}(\tau, p,))$.

Thus the knownness value depends on three parameters: the size of the cell, the number of points inside the cell and the number of points inside the whole tree. Therefore, if the ratio between the number of points contained in a cell and its size does not evolve, its knownness value will decrease.

The insertion of points inside the kd-tree follows this rule: if adding the point to its corresponding leaf l_0 would lead to a number of points greater than ν, then the leaf is split into two leaves l_1 and l_2 of the same size, and the dimension is chosen using a round-robin. The points initially stored in l_0 are attributed to l_1 and l_2 depending on their value.

MRE uses the knownness function to alter the update of the Q-value function. Consider $R_{\max}$ the maximal reward that can be awarded in a single step, it replaces line 8 of Algorithm 13 by:

$$Y = \left\{ \kappa(s, a) \left(r_k + \gamma \max_{a \in A} \tilde{Q_{n-1}}(s'_k, a) \right) + (1 - \kappa(s, a)) \frac{R_{\max}}{1 - \gamma} \mid k \in \{1, \ldots, |\mathcal{D}|\} \right\}$$

This change can be seen as adding a transition to a fictive state containing only self-loop and leading to a maximal reward at every step. This new transition occurs with probability $1 - \kappa(s, a)$.

Computation of the knownness value

The definition of the knownness given by Equation (2.11) leads to the unnatural fact that adding a point in the middle of other points can decrease the knownness of these points. Consider a leaf l_0 with a knownness k_0, then adding a point can result in creating two new leaves l_1 and l_2 with respective knowledge of k_1 and k_2 with $k_0 > k_1$ and $k_0 > k_2$.

We decide to base our knowledge on the ratio between the density of points inside the leaf and the density of points. Thus replacing Eq. (2.11) by Eq. (2.12):

$$\kappa(p) = \min\left(1, \frac{\frac{|\text{points}(\text{leaf}(\tau,p))|}{|\text{leaf}(\tau,p)|}}{\frac{n}{|S\times A|}}\right) \tag{2.12}$$

where n is the total number of points inside the tree. This definition leads to the fact that at anytime, there is at least one leaf with a knownness equal to 1. It is also easy to see that there is at least one leaf with a knownness strictly lower than 1, except if all the cells have the same density.

From knownness tree to knownness forest

In order to increase the smoothness of the knownness function, we decided to aggregate several kd-trees to grow a forest, following the core idea of Extra-Trees [Geurts *and al.* 2006]. However, in order to grow different kd-trees from the same input, the splitting process needs to be stochastic. Therefore, we implemented another splitting scheme based on ExtraTrees.

The new splitting process is as follows: for every dimension, we choose at uniform random a split between the first sample and the last sample. Thus, we ensure that every leaf contains at least one point. Then we use an heuristic to choose the best split.

Once a knownness forest is grown, it is easy to compute the knownness value by averaging the result of all the trees.

Modification of the Q-value update

The Q-value update rule proposed by MRE improves the search speed, however it has a major drawback. Since it only alters the training set used to grow the regression forest, it can only use the knownness information on state action combination which have been tried. Even if a couple (s, a) has a knownness value close to 0, it might have no influence at all on the approximation of the Q-value.

In order to solve this issue, we decided to avoid the modification of the training set creation. In place of modifying those samples, we update the regression forest by applying a post-processing on every of every tree in the regression forest representing the policy. This post-processing is presented in Eq. (2.13). In this equation, l is the original leaf, l' the modified leaf and $c = \text{CENTER}(\mathcal{H}(l))$ is the center of the leaf space.

$$l' = \text{wAvg}\left(l, \mathcal{C}\left(R_{\max}\right), \kappa(c), (1-\kappa(c)\right) \tag{2.13}$$

Since we consider only constant and linear models in leaves, the weighted average of models in Eq. (2.13) can easily be computed, see section 2.1.7.

2.2.4 Semi-online experiments

We evaluated the performance of the combination of MRE and FPF on two different problems. First, we present the experimental results on the *Inverted Pendulum Stabilization* problem and compare them with the results obtained with random exploration. Second, we exhibit the results on the *Inverted Pendulum Swing-Up* problem. Since semi-online learning on robots can be expensive in time and resources, we did not allow for an early phase of parameter tuning and we used simple rules to set parameters for both problems. In both problems, the policy is updated at the end of each episode, in order to ensure that the system is controlled in real-time. In this section, we denote by *rollout* an execution of the MRE algorithm on the problem.

Inverted pendulum stabilization

This problem is exactly the same as defined in Section 2.2.2, but it is used in a context of semi-online reinforcement learning. After each rollout, the policy is updated using all the gathered samples. The results presented in this section represent 10 rollouts of 100 episodes. Each rollout was used to generate 10 different policies, every policy was evaluated by 50 episodes of 3000 steps. Thus, the results concerns a total of 5000 evaluations episodes.

The repartition of reward is presented in Figure 2.10. The reward obtained by the best and worst policy are shown as thin vertical lines, while the average reward is represented by a thick vertical line. Thus, it is easy to see that there is a huge gap between the best and the worst policy. Over this 5000 episodes, the average reward per run was -171, with a minimum of -1207 and a maximum of -128. In the batch mode settings, after the same number of episodes, FPF-PWL obtained an average reward of -172, with a minimal reward of -234 and a maximal reward of -139. While the average reward did not significantly improve, the dispersion of reward has largely increased and in some cases, thus leading to better but also worst policy. While this might be perceived as a weakness, generating several policies from the computed Q-value is computationally cheap. Then, a few episodes might be used to select the best policy. From the density of reward presented in Figure 2.10, it is obvious that by removing the worst 10% of the policies, the average reward would greatly improve.

Another point to keep in mind is the fact that the parameters of FPF have not been optimized for the problem in the MRE setup, while they have been hand-tuned in the Batch setup. Therefore, reaching a comparable performance without any parameter tuning is already an improvement.

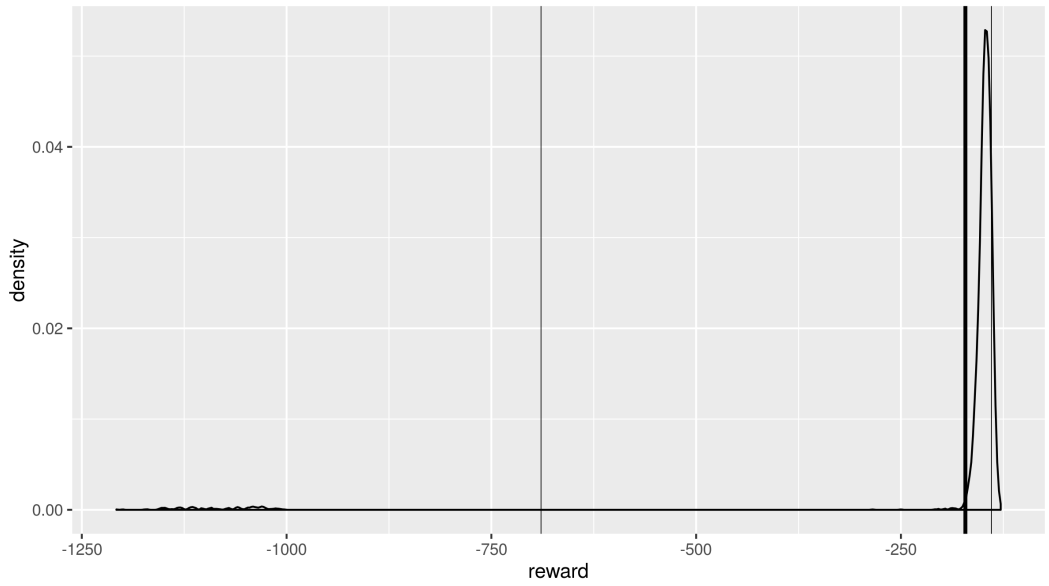


Figure 2.10 – Reward repartition for semi-online learning on Inverted Pendulum Stabilization

Inverted pendulum swing-up

The problem is to control a motor which applies a torque on a pendulum. The goal is to stabilize the pendulum in upward position while it starts pointing down. The main difficulty of this problem is the torque limitation of the motor. It is necessary to start by accumulating energy in the system in order to be able to reach the upward position. The agent can observe the position and the speed of the angular joint and his action is defined by choosing the torque applied by the motor. More details on this problem are provided in Section 1.1.4.

For this problem, instead of using a mathematical model, we decided to use the simulator Gazebo[2] and to control it using ROS[3]. Since these two tools are widely accepted in the robotic community, we believe that exhibiting reinforcement learning experiments based on them can contribute to the democratization of reinforcement learning methods in robotics.

The result presented in this section represent 5 rollouts of 100 episodes. Each rollout was used to generate 10 different policies, every policy was evaluated by 10 episodes of 100 steps. Thus, there is a total of 500 evaluation episodes.

We present the repartition of the reward in Figure 2.11. The average reward is represented by a thick vertical line and the best and worst policies rewards are shown by thin vertical lines. Again, we can notice a large difference between

[2]http://gazebosim.org
[3]http://www.ros.org

the best and the worst policy. We exhibit the trajectory of the best and worst evaluation episode in Figure 2.12. While the worst episode has a cumulated reward of -101, the worst policy has an average reward of -51. According to the repartition of the reward, we can expect that very few policies lead to such unsatisfying results, thus ensuring the reliability of the learning process if multiple policies are generated from the gathered samples and a few episodes are used to discard the worst policy.

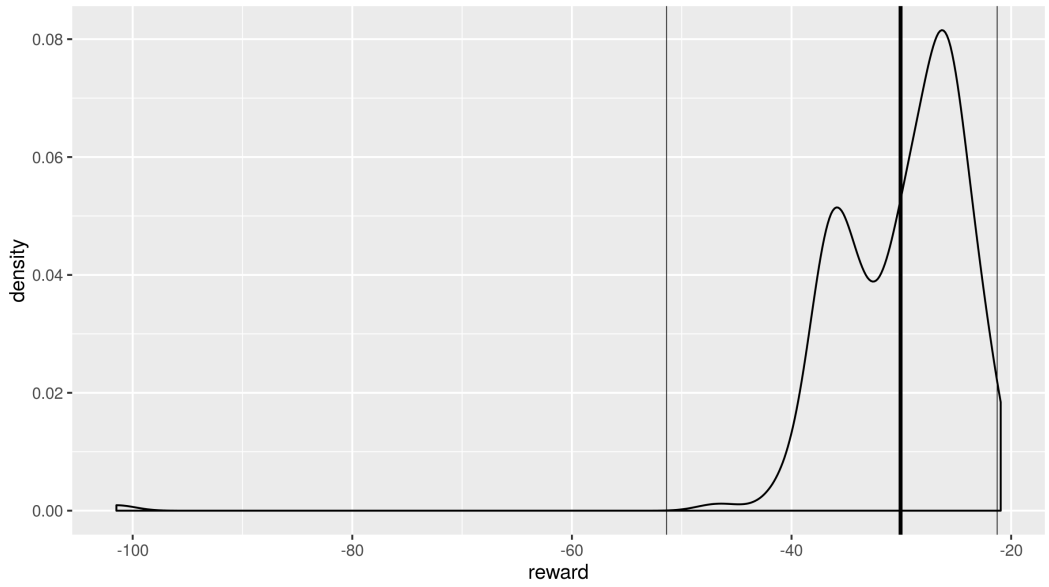


Figure 2.11 – Reward repartition for online learning on Inverted Pendulum Swing-Up

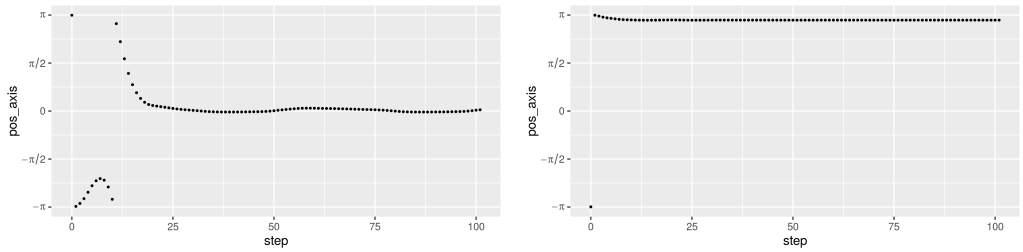


Figure 2.12 – Best and worst episode for Inverted Pendulum Swing-Up

2.2.5 Discussion

Our results show that using FPF does not only allow to drastically reduce the online computational cost, it also tend to outperforms FQI and BAS, especially when the transition function is stochastic as in the Inverted Pendulum Stabilization problem.

Although using piece-wise linear function to represent the Q-value may lead to divergence as mentioned in [Ernst *and al.* 2005], this was not the case for any of the three presented problems. In two of the three presented benchmarks, FPF:PWL yields significantly better results than FPF:PWC and on the last problem, results were similar between the two methods. The possibility of using PWL approximations for the representation of the policy holds in the fact that the approximation process is performed only once. Another advantage is the fact that on two of the problems, the policy function is continuous. However, even when the optimal policy is bang-bang (Car on the hill), using PWL approximation for the policy does not decrease the general performance.

Our experiments on the combination of MRE and FPF showed that we can obtain satisfying results without a parameter-tuning phase. Results also show the strong variability of the generated policies, thus leading to a natural strategy of generating multiple policies and selecting the best in a validation phase.

Although several problems can be solved using model-free learning, there is a major limitations to its use in robotics. It does not make an efficient use of the samples gathered on the robots while acquiring these samples is generally time-consuming. In high dimensional problems, sample efficiency is even more crucial. Moreover, when tackling robotics problems, it is common to use simulators because experimenting directly on the robot is more expensive, slower and often dangerous. Therefore, if there is no available model of the problem, it is generally very helpful to start by building an approximation of the problem.

Separating the learning of the model and the learning of the policy has two major advantages: first, it allows evaluating both separately thus making debugging much easier, second, since models in robotics can be used for different problems, it is useful to dissociate them from the task we are trying to accomplish.

2.3 Random Forest Policy Iteration

Regression Forests Policy Iteration, or RFPI for short, is inspired by the discrete version of *policy iteration*. Both the value function and the policies are represented as *regression forests*. This is a blackbox learning algorithm: it relies on sampling the behavior of the controlled system from various initial states, chosen by the algorithm. Our implementation is limited to CSA-MDPs with homogeneous action spaces.

RFPI uses visited states as a basis to train the value functions and the policies. The list of visited states may be empty at the beginning or it might start with states provided by an external source to guide the learning.

RFPI iterates the following three steps:

1. Performing rollouts to discover new states according to Algorithm 16.
2. Updating the estimation of the value function
3. Updating the policy

Algorithm 16 Performing a rollout from an initial state

```
1: function PERFORMROLLOUTFROMSTATE(MDP,s,π, H)
2:                 ▷ Parameters description:
                       MDP    The model of the CSA-MDP
                       s      The initial state
                       π      The policy
                       H      The maximal horizon for the rollout
3:     states ← {}
4:     n ← 0
5:     status ←⊢
6:     cumulatedReward ← 0
7:     factor ← 1
8:     while n < H ∧ status ≠⊣ do
9:         visitedStates.insert(s)
10:        (s, r, status) ← SAMPLERESULT(MDP, s, π(s))
11:        cumulatedReward ← cumulatedReward + r ∗ factor
12:        factor ← factor ∗ DISCOUNT(MDP)
13:    end while
14:    return (visitedStates, cumulatedReward)
15: end function
```

Initially, the value is considered to be 0 for every state and the initial policy samples actions from a uniform distribution among the whole action space. The algorithm is based on a time budget: steps 1. 2. and 3. are repeated until the time allowed is elapsed. At step 1., a fixed number of rollouts are performed using the function PERFORMROLLOUTFROMSTATE, see Algorithm 16. Step 2. evaluates the expected reward by averaging the rewards returned by the rollouts. The estimation of the value function is updated using the function EXTRATREES, see Algorithm 17. We use piece-wise constant models to approximate the value function, because experiments and literature suggest that using linear-approximators for a discretization of the problem might lead to a divergence toward infinity, see [Ernst *and al.* 2005]. Step 3. updates the policy using Algorithm 18. The complete version of RFPI is described in Algorithm 19.

In order to update the value and the policy, it is required to gather a large number of samples using the blackbox model of the MDP, see Algo-

Algorithm 17 The RFPI update value function

```
1: function RFPIValue(MDP, π, states, n,H)
2:     ▷ Parameters description:
           MDP      The model of the CSA-MDP
           π        The policy which should be used
           states   A list of states used as initial states for the rollouts
           n        The number of rollouts to use
           H        The maximal horizon for the rollout
3:     values ← {}
4:     for all s ∈ states do
5:         rollout ← 0
6:         stateReward ← 0
7:         while rollout < n do
8:             (unused, r) ← performRolloutFromState(MDP, s, π, H)
9:             stateReward = r + stateReward;
10:            rollout+ = 1
11:        end while
12:        values.insert(stateReward / n)
13:    end for
14:    return ExtraTrees(states, values, PWC)                    ▷ Ṽ
15: end function
```

rithm 17 and Algorithm 18. Updating the value function can require up to $|\text{visitedStates}| v_{\text{rollouts}} H$ depending on the presence of terminal states during the rollouts and updating the policy requires $|\text{visitedStates}| a_{\text{rollouts}} n_{\text{actions}}$. Therefore, it is apparent that an efficient simulation of the MDP process is crucial regarding the performances of RFPI.

Considering that the tree produced is balanced, the complexity of function ExtraTrees is $O(n \log(n))$, with n the number of samples, see Section 2.1.5. During the execution of the RFPI algorithm, updates of both the value and the policy are performed after a growing number of rollouts: i.e. they are update after $1, 3, 6, \ldots, \frac{k(k+1)}{2}$ rollouts. This ensures that the number of updates performed is in $O(\sqrt{N})$. Therefore, after N rollouts, the complexity is $O(NH\sqrt{N}\log(NH))$.

Chapter 3 presents the experimental results obtained with RFPI on the ball approach problem in both simulation as well as real-world experiments. The experimental results on the *ball approach problem* are satisfying: RFPI could synthesize a policy which performs much better than both expert policy and policies obtained by other optimization methods. Since our implementation of RFPI does not handle heterogeneous action spaces, we could not apply it to

Algorithm 18 The RFPI update policy function

```
1: function RFPIPolicy(MDP, Ṽ, states, n_actions, n_rollouts)
2:     ▷ Parameters description:
           MDP          The model of the CSA-MDP
           Ṽ            A value approximator used to estimate the value function
           states       A list of states used as initial states for the rollouts
           n_actions    The number of actions to use
           n_rollouts   The number of rollouts to use
3:     bestActions ← {}
4:     for all s ∈ states do
5:         bestScore ← −∞
6:         bestAction ← NULL
7:         candidate ← 0
8:         while candidate < n_actions do
9:             action ← sample(actionSpace(MDP))
10:            score ← 0
11:            rollout ← 0
12:            while rollout < n_rollouts do
13:                (s′, r, status) = sampleResult(MDP, s, action)
14:                score ← score + r
15:                if status =⊢ then    ▷ If status is not terminal, use s′ value
16:                    score ← score + discount(MDP)Ṽ(s′)
17:                end if
18:                rollout ← rollout + 1
19:            end while
20:            if score > bestScore then
21:                bestScore ← score
22:                bestAction ← action
23:            end if
24:            candidate ← candidate + 1
25:        end while
26:        bestActions.insert(bestAction)
27:    end for
28:    return ExtraTrees(states, bestActions, PWL)    ▷ The new policy
29: end function
```

Algorithm 19 Random Forest Policy Iteration

function RFPI(MDP, π, seedStates,v_{rollouts}, n_{actions},a_{rollouts},H)
    ▷ Parameters description:

MDP	The model of the CSA-MDP
seedStates	An expert set of states used to guide RFPI
$v_{\textbf{rollouts}}$	The number of rollouts to use in value update
$n_{\textbf{actions}}$	The number of actions to test in policy update
$a_{\textbf{rollouts}}$	The number of rollouts to use in update policy
H	The maximal horizon for the rollouts

    $\pi \leftarrow \mathcal{U}(\mathcal{H}_A)$ ▷ Initial policy is random
    $\tilde{V} \leftarrow \mathcal{C}([0])$
    visitedStates ← seedStates
    policyId ← 1
    runId ← 0
    while timeRemaining() **do**
        $s \leftarrow$ sampleInitialState(MDP)
        (newStates, r) ← performRolloutFromState(MDP,s,π,H)
        visitedStates.insert(newStates)
        runId ← runId + 1
        if runId = policyId **then**
            $\tilde{V} \leftarrow$ RFPIValue(MDP, π, visitedStates, v_{rollouts}, H)
            $\pi \leftarrow$ RFPIPolicy(MDP, $\tilde{V}$, visitedStates, n_{actions}, a_{rollouts})
            runId ← 0
            policyId++
        end if
    end while
    return π
end function

the *kicker robot problem*.

2.4 Policy Mutation Learner

This section introduces Policy Mutation Learner, PML for short. This algorithm tracks a policy represented by a *regression tree* with linear models in the leaf, see Figure 2.1 for an example. It optimizes the policy by performing local mutations and ensuring that they have a positive impact on the global reward.

One of the main motivations behind the development of PML is the necessity of specific algorithms able to handle efficiently CSA-MDP with het-

erogeneous action spaces. By incorporating the possibility to change the type of action used inside the mutation mechanism, PML can handle models with heterogeneous action spaces.

Experimental results obtained using PML are provided in chapter 4. They show that PML can successfully be used for both improving expert strategy and finding efficient strategies from scratch.

2.4.1 Blackbox optimization

In this section, different methods of blackbox optimization are presented. All of them maximize the expected reward of a blackbox function $\mathcal{B} : I \mapsto \Delta(\mathbb{R})$, i.e. they evaluate $\sup_{i \in I} \mathbb{E}[\mathcal{B}]$. Of course these optimizers can also be used to minimize the expected cost.

There are various types of optimization problems and optimizers, this section presents distinctions between them based on several criteria.

Stochastic and deterministic blackboxes

While in this thesis, the focus is on stochastic blackboxes in which two strictly equivalent inputs can result in different outputs, some blackbox are deterministic.

Given two samples (i_0, o_0) and (i_1, o_1) obtained from a blackbox $\mathcal{B}$, if the blackbox is deterministic, we can directly assume that i_0 is better than i_1 if and only if $o_0 > o_1$. On the other hand if the blackbox is stochastic, more samples are required in order to provide statistical evidence that one of the input has a higher average output. Moreover, while sampling nearby inputs allow to retrieve a local gradient for deterministic problem, a much larger number of samples is required to provide a rough approximation of the local gradient in stochastic problems.

For stochastic problems, sampling multiple times the same input and averaging the outputs allows to have a better approximation of the reward for the given input. However, using this scheme can quickly increase the number of calls to the blackbox function.

Cheap and costly blackbox functions

A key aspect of blackbox optimization is the cost of sampling the blackbox. While this cost might not always be represented as a numerical value, it is commonly accepted to separate blackbox in two different categories: cheap and costly. Cheap blackbox can be called several thousands or even millions of time while costly function are usually called at most a few hundred times.

Since this thesis focuses on offline learning, we generally consider cheap blackbox functions based on models trained on data acquired on the robot. However, if optimization of the function had to be performed online, then the

blackbox function would be considered as costly because learning on the robot includes risk of damaging the robot and requires human supervision.

Limited memory

Optimizers with a limited memory are generally based on using a simple state including information about the current candidate, this candidate might simply be the input of the function or it might also take a more complex form such as a distribution with a covariance matrix. At each step, the optimizer samples the blackbox function to update its candidate using the new samples acquired.

A major advantage of this approach is the fact that, since the memory is limited, the time required to choose the next inputs to sample remains reasonable. However, since they do not use all the samples gathered during the process, they tend to have a lower sample efficiency than methods who memorize all the samples used since the beginning of the optimization. Optimizers with limited memory will typically be used to optimize cheap blackbox functions.

Local and global optimizers

We separate the methods in two different categories: local and global optimization. While local optimization focuses on finding efficiently a local maximum, global optimization aims at providing global maximum over the whole domain of the function.

While limiting the search to a local scope often allows to provide satisfying results much faster than global search, it has two major drawbacks. First, local search is particularly sensitive to initialization of the search, therefore it generally requires human expertise. Second, when multiple local maxima exist for the blackbox function, then local optimizers are likely to converge to sub-optimal inputs.

2.4.2 Main principles

The concept behind PML is to seek to optimize internal parameters of a policy in a similar way to policy gradients algorithms. However it does not require either a symbolic expression of the transition function nor the parametrized policy. Due to the focus on local parameters and the access to the blackbox function, PML can restrict acquisition of samples to states where modifications of the policy are susceptible to improve the average reward. Since PML requires access to a cheap blackbox function, it is not mandatory to approximate the value functions when updating the policy, the value guaranteed by the policy at a given state can simply be estimated by sampling trajectories of the system given by the blackbox function.

In order to keep track of the evolution of the tree, PML keeps track of the number of mutations performed. We note $m_{\text{ID}} \in \mathbb{N}$ the number of mutations performed at anytime.

2.4.3 Memory

As mentioned previously, PML does not store approximation of the Q-value function or the value function. It stores only the states visited during the last evaluation of the policy, the *regression tree* representing the current policy and additional information about its leaves.

We denote $\textsc{visitedStates}(l)$ the states in $\mathcal{H}(l)$, among states visited during the last policy evaluation.

We denote $\textsc{lastMutation}(l)$ the value of m_{ID} at the last mutation of leaf l.

We denote $\textsc{actionId}(l)$ the identifier of the action applied by leaf l, since we used heterogeneous actions, it is mandatory to store this additional information.

2.4.4 Mutations candidates

In the PML algorithm, all the leaves of the current policy are considered as candidates for mutation. However their chances of being selected for a mutation depend on multiple parameters.

Let us consider a tree t, a node $l \in \textsc{nodes}(t)$ and $\alpha \in [1, \infty[$. We define $\textsc{weight}(l)$ by:

$$\textsc{weight}(l, \alpha) = (1 + |\textsc{visitedStates}(l)|)\alpha^{\frac{m_{\text{ID}} - \textsc{lastMutation}(l)}{\textsc{nbLeaves}(t)}}$$

The probability of a leaf l to be selected at mutation m_{ID} is given by Eq. (2.14).

$$p(l) = \frac{\textsc{weight}(l, \alpha)}{\sum_{l' \in \textsc{nodes}(t)} \textsc{weight}(l', \alpha)} \tag{2.14}$$

The parameter α controls the trade-off between time elapsed since last mutation and the number of visit a leaf received on its probability to be selected. If $\alpha = 1$, then the chances for a state of being selected will be proportional to the number of times it has been visited during the last evaluation of the policy. On the other hand, large values of α will lead PML to always update the leaf which has spent the most time without training.

The heuristic PML uses for scoring is based on two observations. First, improving the policy in a frequently visited part of the space is likely to bring significant improvement on the global performances. Second, if mutations always occurs on the same subtree because it is visited more often, there is a

risk of converging to a locally optimal policy, because further improvements would require to mutate leaves from another subtree.

2.4.5 Optimization spaces

Among the key parameters of optimization problems, choosing an appropriate optimization space is crucial for performances. In PML, the aim is to optimize the parameters of constant and linear models used in leaves. We present here the spaces used for constant and linear matrices.

The limits used for constant coefficients are the same as the limit of the parameters, thus ensuring that the spaces for constant coefficients only contains meaningful values. It is denoted $\textsc{constantBounds}(\text{MDP}, i)$ with MDP the model of the CSA-MDP and i the id of the action concerned:

$$\textsc{constantBounds}(\text{MDP}, \text{actionId}) = \textsc{actionSpace}(\text{MDP}, \text{actionId})$$

The bounds of acceptable values for a coefficient of a matrix A representing the linear coefficients of a model is defined by $A_{i,j} \in [-m_{i,j}, m_{i,j}]$ with:

$$m_{i,j} = \frac{\textsc{length}(\textsc{actionSpace}(\text{MDP}, \text{actionId}), i)}{\textsc{length}(\textsc{stateSpace}(\text{MDP}), j)}$$

where $\textsc{length}(\mathcal{H}, i)$ is the length of the hyperrectangle $\mathcal{H}$ over dimension i.

We denote the space of acceptable matrix for linear coefficients for action i by $\textsc{linearBounds}(\text{MDP}, i)$. It contains all the matrices of the appropriate size with values in the range specified above.

2.4.6 Mutation types

The first step of a mutation is to choose the type of mutation that will be performed. There are two different types of mutations: *refine mutations* and *split mutations*. The type of the mutation is chosen randomly according to the parameters $p_{\text{split}} \in [0, 1]$ that defines the probability of performing a *split mutation*.

Both mutations rely on the evaluateFA functions to optimize internal parameters. This function allows to evaluate the expected value of the total reward received for the initial states provided. It is described in Algorithm 20.

All the mutation are submitted to a validation process named validateFA ensuring that modifications to the current approximator improve the performances both locally and globally. This procedure is described in Algorithm 21.

Refine mutations

The goal of a *refine mutation* is to try to find a more suitable local model for the space concerned by the leaf. A *refine mutation* follows algorithm 22.

Algorithm 20 Evaluation of new function approximators in PML

1: **function** EVALUATEFA(MDP,π, l,l',S_{initials},H)
2: ▷ Parameters description:

MDP	The blackbox model of the MDP
π	The policy before modification
l	The function approximator to replace
l'	The function approximator replacing l
S_{initials}	The set of initial states used for training, based on visited states during last evaluation
H	The horizon until which evaluation is performed

3: $l_{\text{backup}} \leftarrow l$
4: $l \leftarrow l'$
5: $r \leftarrow 0$
6: **for all** $s \in S_{\text{initials}}$ **do**
7: $(\text{unused}, r') \leftarrow$ PERFORMROLLOUTFROMSTATE(MDP, s, π, H)
8: $r \leftarrow r + r'$
9: **end for**
10: $l \leftarrow l_{\text{backup}}$ ▷ Restore old approximator
11: **return** r
12: **end function**

Finding the best argument (A, B) at line 8 of Algorithm 22 is a task performed by a blackbox optimizer.

Split mutation

The aim of split mutations is to prepare an appropriate structure for further mutations of function approximators, therefore, it is not required that they bring an immediate improvement to the expected reward for the policy. However, performing appropriate splits is still important because split mutations shape the main structure of the policy and allow discontinuities.

The procedure we use to perform *split mutations* is described at Algorithm 23. For each dimension of the state space, the algorithm tries to find the most appropriate split using a blackbox optimizer to find the best parameters at line 9.

While *refine mutations* which do not lead to both local and global improvements are simply refused, the mechanism for *split mutations* is different. If the validation test is passed, the new function approximator is accepted. Otherwise, the split is conserved, but the two children are replaced with the original model. This ensures that even if the *split mutation* does not result in

Algorithm 21 Validation of new function approximators in PML

function VALIDATEFA(MDP,π, l,l',H,v_{rollouts})
▷ Parameters description:

MDP	The blackbox model of the MDP
π	The policy before modification
l	The leaf to replace
l'	The function approximator replacing l
H	The horizon until which evaluation is performed
$v_{\textbf{rollouts}}$	The number of rollouts used to estimate global reward

$S_{\text{init}} \leftarrow$ VISITEDSTATES(l)
$S_{\text{glob}} \leftarrow \{\}$
while $|S_{\text{glob}}| < v_{\text{rollouts}}$ **do**
  $S_{\text{glob}} \leftarrow S_{\text{glob}} \cup$ SAMPLEINITIALSTATE(MDP))
end while
$r_{\text{loc}} \leftarrow$ EVALUATEFA(MDP, $\pi, l, l, S_{\text{init}}, H$)
$r'_{\text{loc}} \leftarrow$ EVALUATEFA(MDP, $\pi, l, l', S_{\text{init}}, H$)
$r_{\text{glob}} \leftarrow$ EVALUATEFA(MDP, $\pi, l, l, S_{\text{glob}}, H$)
$r'_{\text{glob}} \leftarrow$ EVALUATEFA(MDP, $\pi, l, l', S_{\text{glob}}, H$)
return $r'_{\text{loc}} > r_{\text{loc}} \wedge r'_{\text{glob}} > r_{\text{glob}}$
end function

improvement, it will still increases the number of leaves, thus allowing further refinement to develop more complex policies.

2.4.7 Core of the algorithm

The complete definition of PML is presented in Algorithm 24. At every step of the algorithm, the states stored in memory are updated in order to reflect the current policy. A leaf is chosen randomly according to Eq. (2.14). The type of mutation to be used is chosen randomly according to the parameter p_{split}. Finally, the algorithm proceed to the mutation.

2.4.8 Parameters and initial knowledge

The parameters of PML are the following:

allowedTime	The available time for improving the policy.
H	The horizon at which the problem is solved.

Algorithm 22 The refine mutation for PML

function REFINEMUTATION(MDP,π,l,H,v_{rollouts})
▷ Parameters description:

MDP	The blackbox model of the MDP
π	The policy before mutation
l	The leaf on which mutation should be performed
H	The horizon until which evaluation is performed
$v_{\textbf{rollouts}}$	The number of rollouts used to estimate global reward

$S_{\text{init}} \leftarrow$ VISITEDSTATES(l)
actionId $\leftarrow$ randomSample($\{1, \ldots,$ NBACTIONSPACES(MDP)$\}$)
$\text{space}_A \leftarrow$ LINEARBOUNDS(MDP, actionId)
$\text{space}_B \leftarrow$ CONSTANTBOUNDS(MDP, actionId)
$c \leftarrow$ CENTER($\mathcal{H}(l)$)
$(A, B) \leftarrow \underset{\substack{A \in \text{space}_A \\ B \in \text{space}_B}}{\arg\max}$ EVALUATEFA(MDP, $\pi, l, \mathcal{L}(A, B + Ac), S_{\text{init}} H$)
$l' \leftarrow \mathcal{L}(A, B + Ac)$
if VALIDATEFA(MDP, $\pi, l, l', H, v_{\text{rollouts}}$) **then**
$l \leftarrow l'$
ACTIONID(l) $\leftarrow$ actionId
end if
end function

nbEvaluationRollouts	The number of rollouts used for validation in order to have an accurate estimation of the average reward.
trainingEvaluations	The maximal number of runs used for evaluating the average reward when solving inner blackbox optimization problems during a mutation.
p_{split}	The probability of performing a split mutation at each step. It should always be lower than $\frac{1}{\lvert A \rvert}$, in order to ensure that enough different actions are tested between split mutations.
evaluationsRatio	This parameter allows to control the ratio between the number of runs simulated for solving blackbox optimizer and for validation.
α	The age basis used to compute the weights of the different candidates for mutation.

Algorithm 23 The split mutation for PML

function SPLITMUTATION(MDP,π,l,H,v_{rollouts})
▷ Parameters description:

MDP	The blackbox model of the MDP
π	The policy before mutation
l	The leaf on which mutation should be performed
H	The horizon until which evaluation is performed
$v_{\textbf{rollouts}}$	The number of rollouts used to estimate global reward

$r_{\max} \leftarrow -\infty$
$t_{\text{best}} \leftarrow$ NULL
$S \leftarrow$ STATESPACE(MDP)
for $d \in \{0, \ldots, |S|\}$ **do**
  $\text{space}_C \leftarrow$ CONSTANTBOUNDS(MDP, actionId)
  $\text{space}_v \leftarrow$ DIM($\mathcal{H}(l), d$) ▷ Select limits for dim d
  $t \leftarrow \underset{\substack{c_1 \in \text{space}_C \\ c_2 \in \text{space}_C \\ v \in \text{space}_v}}{\arg\max}$ EVALUATEFA(MDP, $\pi, l, \mathcal{T}(d, v, c_1, c_2), S_{\text{init}}, H$)
  $r \leftarrow$ EVALUATEFA(MDP, $\pi, l, t, S_{\text{init}}, H$)
  if $r > r_{\max}$ **then**
    $r_{\max} \leftarrow r$
    $t_{\text{best}} \leftarrow t$
  end if
end for
if VALIDATEFA(MDP, $\pi, l, t_{\text{best}}, H, v_{\text{rollouts}}$) **then**
  $l \leftarrow t_{\text{best}}$
else
  $d \leftarrow$ SPLITDIM(t)
  $v \leftarrow$ SPLITVAL(t)
  $l \leftarrow \mathcal{T}(d, v, l, l)$
end if
end function

Additionally to parameters, it is possible to provide initial knowledge to PML by specifying an initial policy under the form of a tree of local models. While this is not mandatory, even a simple approximate guess can improve the performance of PML by ensuring that the initial structure is adapted. This aspect is discussed in detail in section 4.2.2. Thus, PML can also be used for improving expert policies.

Algorithm 24 Policy Mutation Learner algorithm

function PML(MDP,π,H,v_{rollouts},p_{split})
▷ Parameters description:

MDP	The blackbox model of the MDP
π	A regression tree representing the policy
H	The horizon until which evaluation is performed
$v_{\textbf{rollouts}}$	The number of rollouts used to estimate global reward
$p_{\textbf{split}}$	The probability of performing a split at each mutation

    $\pi \leftarrow$ getInitialPolicy()
    $m_{\text{ID}} \leftarrow 0$
    while isTimeRemaining() **do**
        $S_{\text{visited}} \leftarrow \{\}$
        while $|S_{\text{visited}}| < v_{\text{rollouts}}$ **do**
            $S_{\text{visited}} \leftarrow S_{\text{visited}} \cup$ SAMPLEINITIALSTATE(MDP)
        end while
        use S_{visited} to update VISITEDSTATES(l) for all leaves of π
        $l \leftarrow$ a leaf chosen randomly according to Eq. (2.14)
        if SAMPLE($\mathcal{U}\left(\begin{bmatrix}0 & 1\end{bmatrix}\right)$) $< p_{\text{split}}$ **then**
            SPLITMUTATION(MDP, π, l, H, v_{rollouts})
        else
            REFINEMUTATION(MDP, π, l, H, v_{rollouts})
        end if
        $m_{\text{ID}} \leftarrow m_{\text{ID}} + 1$
    end while
end function

2.4.9 Discussion

Among the main weaknesses of PML, we identified that using only orthogonal splits require very deep trees to model policies where the action depend on an inequality with several dimensions involved. Modifications of the algorithm allowing to model other form of splits could bring a substantial improvement by reducing the number of leaves requiring refinement. Moreover, expert information can also contain information about symmetries in the policies. This kind of expert information could easily be provided if the policies were described as eXtended Algebraic Decision Diagrams, XADD for short, see [Sanner *and al.* 2012]. Moving from regression trees to XADD would allow to represent policies in a more compact way and avoid the necessity of training and storing similar models in different leaves.

While Chapter 4 presents satisfying results based on PML, this algorithm does not use information from previous mutations to guide the optimization of parameters. Developing a mechanism allowing to acquire global knowledge through mutation could strongly help to avoid performing similar operations during different mutations.

Finally, one of the most important problem of the proposed version of PML is the fact that it only perform mutations on leaves and can only expand the tree. By adding a mechanism to remove node from the trees, it could be possible to reuse policies produced by PML as seeds for other executions of PML. This would enable procedures where the main structure of the tree is acquired by training a tree on a cheap approximation of the problem, then the policy would be refined using a more expensive but more accurate model. The interest of this scheme is highlighted by the experiments presented in section 4.2.1. Moreover, this would allow to simply adapt old strategies when changing parameters of a problem, thus resulting in a higher flexibility.

2.5 Online planning

While learning policies offline allows taking decision at a low computational cost, online planning can lead to efficient solutions because it does not need exploring the entire state space. In order to reduce the computational burden of online planning, we decided to optimize only the next action, relying on the policy learned offline for following steps.

Since we suppose access to a rollback policy π, we can easily average the Q-value of a couple state-action over multiple samples. We denote this process of performing online evaluation ONLINEEVAL. It is described in Algorithm 25.

Based on our evaluation of the Q-value, we can easily express one step online-planning as a blackbox optimization problem. The algorithm we propose is named ONLINEPLANNING and is presented in Algorithm 26. If the optimization fails to find an action yielding a higher expected reward than $\pi(s)$ with π the rollback policy and s the current state, then the online planning return $\pi(s)$. This mechanism ensures that the online planning may only improve the expected reward with respect to π, given that the number of rollouts for validation is large enough.

2.6 Open-source contributions

This section presents the open-source contributions of this thesis. The contribution includes an implementation in `C++` of all the algorithms developed in this thesis and other algorithms which are beyond the scope of this thesis.

The main architecture of the contributions is presented in the graph presented in Fig. 2.13. Every node is a separate *ROS* package, rectangular nodes

Algorithm 25 Online evaluation of the Q-value

function ONLINEEVAL(MDP,s,a,π,H,n)
▷ Parameters description:

MDP	The blackbox model of the MDP
s	The initial state
a	The first action to take
π	A regression tree representing the policy
H	The horizon until which evaluation is performed
n	The number of rollouts used for evaluation

$r_{\text{tot}} \leftarrow 0$
$i \leftarrow 1$
while $i \leq n$ **do**
  $(s', r, \text{status}) \leftarrow$ SAMPLERESULT(MDP, s, a)
  $r_{\text{tot}} \leftarrow r_{\text{tot}} + r$
  if status $\neq \dashv$ **then**
    $(\ldots, r) \leftarrow$ PERFORMROLLOUTFROMSTATE(MDP, $s', \pi, H-1$)
    $r_{\text{tot}} \leftarrow r_{\text{tot}} + r$
  end if
end while
return $\frac{r_{\text{tot}}}{n}$
end function

with thick borders were developed as a part of this thesis, and elliptic nodes with thin borders are dependencies to external libraries. Arrows between nodes denotes dependencies, dashed arrows denotes optional dependencies. Note that *rosban_ control* has additional dependencies not listed in the graph to improve readability. However, this has a reduced impact since it is an optional dependency for interface with *Gazebo*[4].

If we ignore interface with *Gazebo*, the only required packages from *ROS* are *catkin* and *cmake_ modules*. Those two packages are lightweight and independent from *ROS*. Therefore it is not necessary to installation *ROS* to use the *csa_ mdp_ experiments* package.

Note that we use *Eigen*[5] for all applications of linear algebra, it is a widely spread library which is also used by *ROS*.

The open-source packages have been developed with a specific focus on genericity. We used the concept of *factory* and used it along with the *TinyXML*[6] library to be able to load the configuration of problems, solvers and policies

[4]See http://gazebosim.org/
[5]See http://eigen.tuxfamily.org/
[6]See https://sourceforge.net/projects/tinyxml/

Algorithm 26 A simple online planning algorithm

function ONLINEPLANNING(MDP,π,s,H,n_{eval},n_{valid})
▷ Parameters description:

MDP	The blackbox model of the MDP
π	A regression tree representing the policy
s	The current state
H	The horizon until which evaluation is performed
$n_{\textbf{eval}}$	The number of rollouts used for evaluation
$n_{\textbf{valid}}$	The number of rollouts used for validation

    $a_{\text{best}} \leftarrow \pi(s)$
    $r_{\text{best}} \leftarrow$ ONLINEEVAL(MDP, s, a_{best}, π, H, n_{valid})
    for actionId $\in \{1, \ldots,$ NBACTIONSPACES(MDP)$\}$ **do**
        $A \leftarrow$ ACTIONSPACE(MDP, actionId)
        $a \leftarrow \arg\max_{a \in A}$ ONLINEEVAL(MDP, s, a, π, H, n_{eval})
        $r \leftarrow$ ONLINEEVAL(MDP, s, a, π, H, n_{valid})
        if $r > r_{\text{best}}$ **then**
            $r_{\text{best}} \leftarrow r$
            $a_{\text{best}} \leftarrow a$
        end if
    end for
    return a_{best}
end function

directly from files. Since the xml representation is not suited for large trees, some objects can be written directly in binary files, e.g. regression forests.

We provide here an alphabetically ordered list of all the packages developed during this thesis[7]. The number of lines of each package is shown in Table 2.3.

csa_mdp_experiments
This package contains the implementation of all the problems presented through this thesis. It also contains the source code of expert policies, and programs to launch learning experiments based on the content of configuration files.

rosban_bbo
This package defines the interfaces for blackbox optimizers and implement the three optimizers described in Appendix A: Simulated Annealing, Cross Entropy and CMA-ES. Note that CMA-ES imple-

[7] All packages can be found at `https://www.github.com/rhoban/<pkgname>`.

mentation relies on the external dependency *libcmaes*[8].

rosban_control
This package provides an easy communication with the effectors of a robot simulated in Gazebo and linked to ROS.

rosban_csa_mdp
This package contains the core of the modeling of CSA-MDP. It defines the interface of problems and policies. It also includes all the learning algorithms used in this thesis as well as an implementation of kd-trees, necessary for MRE.

rosban_fa
This package contains *function approximators* and *function approximators trainers*, i.e regression algorithms. By defining a common interface for all regression problems, it allows changing the type of regression or the type of approximator used in algorithms simply by modifying a configuration file. Note that this package can also use Gaussian processes as *function approximators*.

rosban_gp
This package implements *Gaussian processes* based on [Rasmussen 2006]. While *Gaussian processes* are not discussed in this thesis, this package can be used for *function approximators* and could be used in *rosban_bbo* for performing Bayesian optimization.

rosban_random
This simple package contains useful functions to sample elements from various types of distribution.

rosban_regression_forests
This package implements *regression trees* and *regression forests*. It also implements ExtraTrees and all the algorithms manipulating *regression trees* described in this thesis, e.g. projection or pruning. If *rosban_viewer* is present, then a viewer based on *SFML*[9] is used to help visualizing function approximators.

rosban_utils
This package contains various useful tools about serialization and a template definition of the factory concept. It also contains tools making the parallelization of stochastic functions much easier.

[8] see https://github.com/beniz/libcmaes
[9] See https://www.sfml-dev.org/

rosban_viewer
: This package acts as a wrapper for *SFML* and include a `Viewer` class which already contains basic element such as moving a camera inside a 3-dimensional space.

Table 2.3 – Number of lines per package

Package name	lines
`csa_mdp_experiments`	9222
`rosban_bbo`	1038
`rosban_control`	121
`rosban_csa_mdp`	7814
`rosban_fa`	4446
`rosban_geometry`	1468
`rosban_gp`	3141
`rosban_random`	686
`rosban_regression_forests`	4040
`rosban_utils`	1822
`rosban_viewer`	457
Total	**34255**

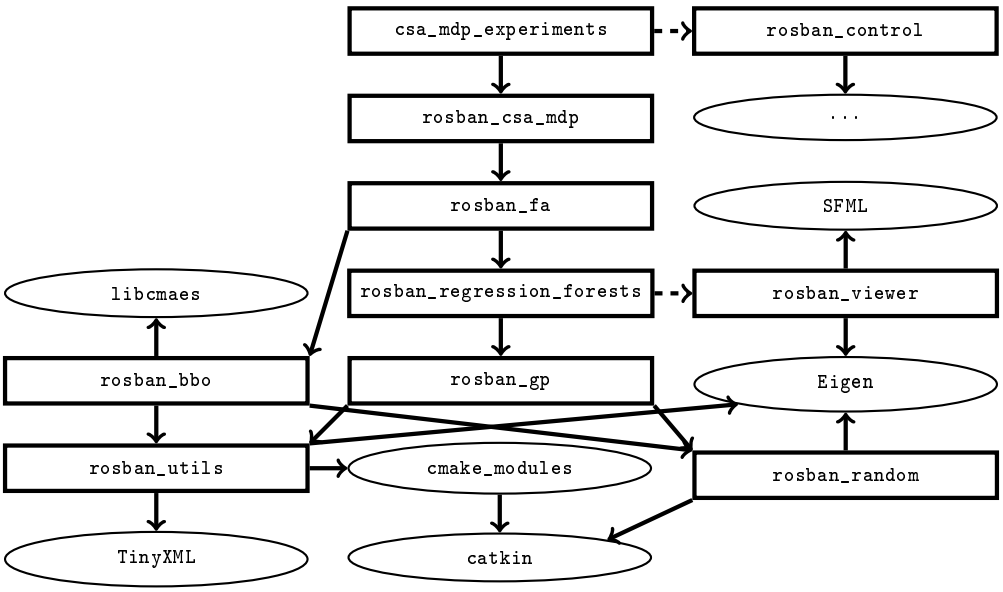


Figure 2.13 – The structure of open-source contributions

Chapter 3

Controlling a humanoid walk engine

During the RoboCup competition, humanoid robots have to be able to control their walk engine in order to reach a position from which they are able to kick the ball towards a desired direction. This problem is complex due to several aspects: the motion of the robot cannot be accurately predicted, observations of the ball position are noisy, the control of the walk engine needs to be smoothed in order to avoid destabilizing the robot and finally, the robot needs to avoid collision with the ball while still minimizing the time required to reach the desired position. From our experience at previous RoboCup, approaching the ball quickly is a key element for victory since it often allows performing the kick before opponent robots reach the ball, thus blocking the path.

In this chapter, experimental results on the ball approach problem are presented and discussed, see section 1.2 for a detailed description of the problem. Those results were published in [Hofer and Rouxel 2017] and were obtained with the collaboration of Quentin Rouxel. While learning a predictive motion model was discussed in 1.2.6, this chapter focuses on computing efficient approach strategies for a given predictive motion model.

Experimental results show that optimization based on RFPI strongly outperforms heuristics we used to win the RoboCup 2016. RFPI also outperforms other optimization methods in both simulation and real-world setups

3.1 Problem setup

Three different policies were experimented in both simulation as well as real world. The simulator used for training and evaluation purposes uses the full linear predictive motion model based on the experimental results presented in section 1.2.6.

Winner2016: The policy used by the Rhoban team to win RoboCup 2016

competition. A simple description is provided in Algorithm 27.

CMA-ES: This policy has the same structure as Winner2016, but all the parameters have been optimized based on millions of simulations of the motion predictive model using the black-box optimization algorithm CMA-ES.

RFPI: This policy is represented as a regression forest. It is the result of a few hours of offline training on the problem. It has no information about the problem but a black-box access to the transition and the reward functions. In order to make sure that the samples were gathered inside the kick area, policies were trained using a seed of 25 trajectories generated by Winner2016.

3.2 Theoretical performances

All the trained policies were evaluated on both problems, HA and ANHA. The evaluation is performed by measuring the average costs on 10'000 rollouts for each modality. The maximal number of steps in a run was set to 50. The simulation results are presented in Table 3.1. All the source code used for simulation experiments is open-source and developed as a set of ROS packages freely available[1].

Table 3.1 – Average costs for the different policies in simulation

	Winner2016	CMA-ES	RFPI
HA	31.84	14.90	11.88
ANHA	44.12	36.18	15.97

First of all, note that on the HA, CMA-ES strongly outperforms Winner2016. This highlights the interest of building a predictive model and using it to optimize the parameters of a policy. By using millions of simulations and advanced optimization methods, CMA-ES is able to divide by more than two the average time to reach the ball position. Our algorithm goes even further and reduces the required time by an additional 20 percent while it has no prior information about the shape of the policy, except a set of visited states by Winner2016.

As we expected, Winner2016 does not perform well at all on ANHA. It has a cost of 44.12 in average, while the maximal cost when avoiding collisions with the ball is 50. Automated tuning based on CMA-ES reduces the average

[1]Source code (C++) available at: https://bitbucket.org/account/user/rhoban/projects/ROS

Algorithm 27 Overview of the expert navigation algorithm

```
1: state = Far
2: while not in kick range do
3:     ball = getRelativeBallPosition()
4:     orientation = getRelativeTargetOrientation()
5:     if state == Far then
6:         Make forward step to go to the ball
7:         Make turn step to align with the ball
8:         if ball.distance is close enough then
9:             state = Rotate
10:        end if
11:    else if state == Rotate then
12:        Make forward step to stay at fixed ball distance
13:        Make lateral step to turn around the ball
14:        Make turn step to stay aligned with the ball
15:        if ball.angle and orientation are aligned then
16:            state = Near
17:        end if
18:    else if state == Near then
19:        Make small forward step to reach kick distance
20:        Make lateral step to keep the ball centered ahead
21:        Make turn step to keep the ball aligned
22:        if |ball.y| lateral position is too far then
23:            state = Rotate
24:        end if
25:    end if
26:    if ball.distance is too far then
27:        state = Far
28:    end if
29: end while
```

cost by around 20 percent. RFPI exhibits a strong flexibility, since it divides by more than two the expected value with respect to the CMA-ES policy. Moreover, RFPI achieves similar performances on ANHA as CMA-ES policy on HA, while the task is much harder.

3.3 Real-world performances

3.3.1 Experimental Conditions

During this experiment, the robot is placed on artificial grass. An official white ball is used as navigation target and the target orientation is set toward the

goal posts. A specific vision pipeline has been implemented to detect and track the ball at 25 Hz. The Cartesian position of the ball with respect to the robot location is obtained by using the model of both the camera and the robot's kinematic. Finally, the position of the ball is filtered through a low pass filter. An example of the experimental setup used is shown in Figure 3.1.

Figure 3.1 – The experimental conditions for the ball approach problem

The recognition and the discrimination between the white ball and the white and round goal posts is a difficult task often leading to false positives. To ease the experiment, the localization module used during robotic competitions has been disabled. The initial kick target orientation is provided manually and is then tracked by integration of the inertial measurement unit.

3.3.2 Experimental Method

Winner2016, CMA-ES policies and RFPI policies are all tested in real soccer conditions for both HA and ANHA. For each test, a total of 12 approaches are run, totaling 72 different approaches. The Cartesian product of the following initial states is performed:

- The ball is put either at 1 m or 0.5 m.
- The ball is put either in front of the robot, on its left, on its right.
- The initial kick target is either 0°or 180°.

For each run, the time required for the robot to stabilize inside the kicking area is recorded.

3.3.3 Results

Average time for all the trajectories depending on the problem type and the policy used are shown in Table 3.2. Even if the quantity of data collected is too small to have an accurate evaluation of the difference of performances among policies, the general trend is similar to the one obtained in simulation. The method tuned by CMA-ES outperforms Winner2016 and RFPI outperforms both.

Table 3.2 – Average time in seconds before kicking the ball

	Winner2016	CMA-ES	RFPI
HA	19.98	13.72	11.45
ANHA	48.14	25.69	18.81

A representation of several trajectories perceived by the robot is given at Fig. 3.2. All these trajectories are directly extracted from the internal representation of the robot[2]. Here, the arrows represent the robot pose at each walk cycle. They all depict the same initial situation, solved for both HA and ANHA, with each of the proposed policies. It can be seen that although the robot started at a distance of 1.0 m of the ball, it initially believes that the distance is around 1.3 m. This type of error is the result of an accumulation of errors in the measurements of the joints, combined to some of the parts bending due to the frequent falls of the robot. The adaptability of the proposed method with respect to the robot constraints can easily be seen by comparing the two trajectories observed for the RFPI policy.

3.4 Discussion

In this chapter, we presented how to train a predictive motion model for a humanoid robot and how cross-validation can help to choose the appropriate type of model. We further presented results showing a major improvement with respect to the policy used at RoboCup 2016. Results were presented in both simulation and real world experiments.

Further development of the presented method can focus on improving the process of training the predictive model. While the current process is easy to use, it still requires major human supervision to launch experiments and measure the displacement of the robots. Moreover, the noise model was chosen manually while it could be extracted from the measurements using a Bayesian method such as marginal likelihood maximization, see [Roy and Thrun 1999]. By modeling the measurement noise properly, we could simply have the robot

[2]A video showing these trajectories on Sigmaban robot is available at: https://youtu.be/PNA-rpNKfsY

wandering on the field, measuring its displacement autonomously based on visual information and then learning its motion model without any human intervention.

While the approach problem we considered has only the position of the ball and the velocity of the robot in the state space, it would be interesting to add the velocity of the ball in order to perform kicking motion from a rolling ball. Achieving such a performance would require a real-time vision system able to track the position of the ball with enough accuracy to estimate its speed. It would also be necessary to learn a model of the evolution of the ball speed according to the artificial turf. During the RoboCup 2017, we noticed that depending on the side to which our robots were kicking, the distance traveled by the ball could double due to the inclination of the blades of the grass. The interest of this feature has also been increased since we recently developed the possibility to perform passes between robots.

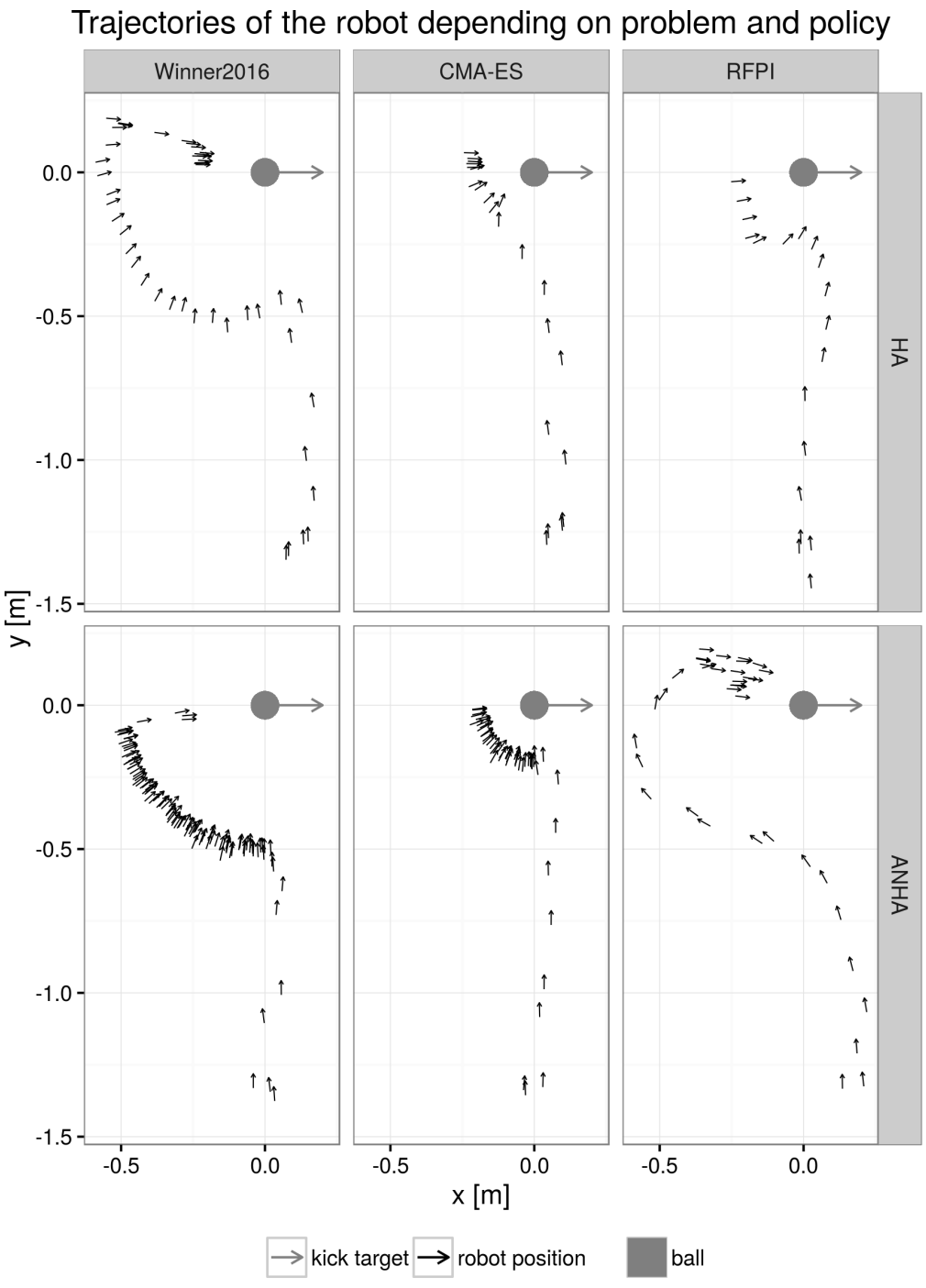


Figure 3.2 – Examples of real world trajectories with different policies

Chapter 4

Kick decisions for cooperation between robots

This chapter presents experimental results obtained on the *kicker robot problem*, see 1.3. Since the modeling of the problem involves several different actions with different parameter spaces, all the offline policies are obtained using PML, see 2.4.

High-level decision modules have to be designed in order to be able to incorporate modifications of the model easily because during the development of robots, the performance of both perception and actuation may improve. Using a learning procedure flexible with respect to the modification of the model also allows assessing the impact of new features on the global performances of the robot. For example, we show that adding to the robot the ability to perform lateral kicks has few impact in the context of the *kicker robot problem*.

While this chapter focuses on PML, it also present results based on online-planning, see Section 2.5. An experimental comparison of the approaches on the *kicker robot problem* is presented in Section 4.2.3.

4.1 Problem setup

We present here specific configurations of the offline solvers and online planners used for the experiments in this chapter.

4.1.1 Offline solvers

For the experiments presented in this chapter, we used PML as the offline learning algorithm. It was not possible to compare its performances with RFPI because the problem involves several distinct actions with their own parameters spaces.

We decided to use a composite optimizer to optimize blackbox functions during mutations. This composite optimizer divides the available budget

between three optimizers: simulated annealing, CMA-ES and cross-entropy. Once the budget is consumed, every optimizer proposes its best candidate, all the candidates are evaluated and only the best one is chosen.

We decided to use this type of composite optimizer because it provided the best results during early experimentation phases. Our understanding of this fact is rather simple, since PML requires solving of various different problems, there is a strong benefit of using different methods. While CE and CMA-ES are able to converge quickly toward a local maximum, SA performs a global search and often provides satisfying solution when CE and CMA-ES are stuck.

Initial seed

Since PML allows providing a seed to guide the search, we experimented the performances of the algorithm with or without providing an expert strategy as a seed. While it is pretty simple, the expert strategy provides a rudimentary policy allowing to center the ball and avoid the goalie.

In the expert policy, the choice of the action is based only on the ball position. This policy is symmetrical and separates the field in 6 different zones with 4 different types:

Defense The ball is simply kicked towards the center of the opponent goal with a powerful kick.

Preparation In this zone, the robot cannot ensure to score in a single kick, therefore it performs a small kick, hoping that it will be able to score a goal at the next kick.

Aisle Center the ball slightly behind. There are two aisle, left and right.

Finish Try to kick between the goalie and the the closest post. Depending on the ball position, the robot will try to kick to the left or the right of the goalie.

A visual description of the policy is shown in figure 4.1, the types of areas are represented by different shades of gray. Two examples of planned kicks for each zone are represented by black arrows. The policy can be represented as a tree with 6 leaves and where every leaf contains a single action. Using an expert kick decision model such as *Finisher* was crucial to design a simple expert policy able to avoid the goalkeeper. Because specifying only the direction of the kick in the field referential would have required a much finer discretization of the space to achieve similar results.

4.1.2 Online planner

All the online planning experiments run in this chapter uses onlinePlanning, see Algorithm 26. Due to the limited number of samples available during online

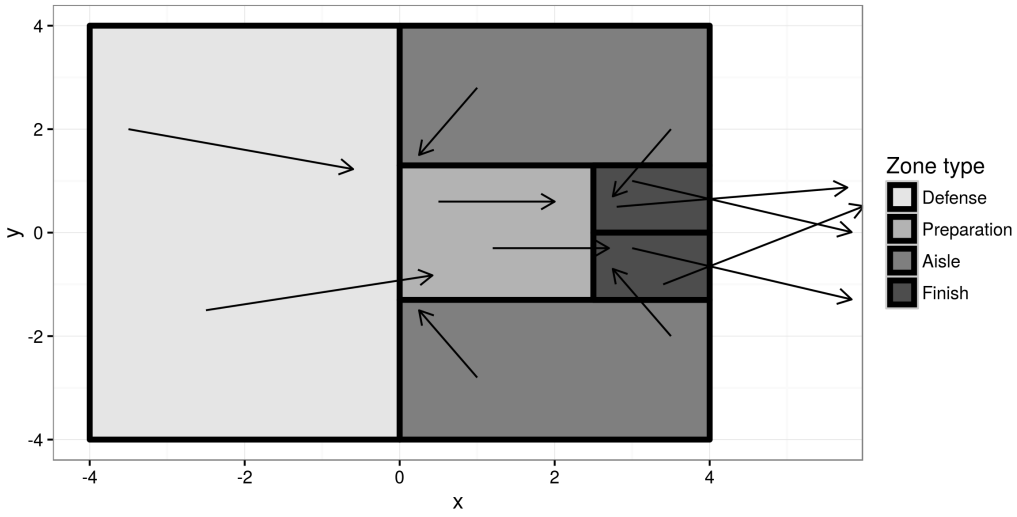


Figure 4.1 – The expert policy used for the kick controller problem

planning, we decided to use the simulated annealing optimizer, see section A.3. We used a simple linear profile for temperature. And the four final parameters of the online planner were:

Temperature Parameter of the simulated annealing algorithm used to establish the trade-off between exploration and exploitation. It was set to 20 for all the experiments.

EvalRollouts The number of rollouts used for estimating the average reward corresponding to an action during the optimization phase.

MaxEvals The number of parameters values allowed to test during the optimization phase. This number is divided among different actions.

ValidationRollouts The number of rollouts used to compute the average reward of each of the final candidates during the evaluation phase. If this number is too low, there is a significant risk of performing worse than the initial policy.

In order to balance the execution time between optimization and validation, we ensured that the number of rollouts involved in validation represents 50 percents of the total number of rollouts, optimization included.

4.2 Results

In this section we present various experimental results we obtained in simulation on the *kicker robot* problem. While this problem specifies a reward function with negative rewards, we present the results as cost, in order to present positive values.

4.2.1 Using surrogate models for inner simulations

Table 4.1 – Average cost received after training for various surrogate models

Training problem	nb robots	Training cost	Validation cost	Mutations
Simul	1	110.43	110.43	1496.4
	2	109.84	109.84	715.7
Speed	1	106.15	103.86	4626.6
	2	94.04	102.88	4090.8

In order to evaluate the impact of using different surrogate models for the displacement of the robots, we trained policies with two types of approach cost, as described in section 1.3.3. For every type of training, we used 50 different training each one using one hour of computation using 20 threads. All the policies training have been seeded using the initial seed described in 4.1.1. The only difference in the configuration of PML is a reduction of the number of samples allowed for optimization in the *Simul* case, due to the important difference of execution time. Despite this modification, the number of iterations performed for *Simul* is still significantly lower than the number of iterations performed for *Speed*.

Evaluation was performed using 1000 rollouts per policy, therefore the results presented in table 4.1 uses 50'000 runs to average the scores. All the policies have been evaluated using both, the problem used for their training and the *Simul* problem which is closer from real-world conditions.

The first glance at table 4.1 shows that *Speed* strongly outperforms *Simul*. It highlights the fact that, when using surrogate models to train offline policies, it is crucial to take into account the trade-off between computational cost and accuracy.

While one and two robots problems yield similar performances during training for *Simul*, there is a significant reduction of the average cost when using two robots and the *Speed* approximation. According to our understanding, this is the result of two different factors: first, *Speed* training has access to more iterations because simulation of the blackbox is less expensive, second, approach cost is lower with *Speed* because there is no noise involved in robots displacement. On the other hand, policies for the two robots problems were supposed to yield better results for all problems when compared with the single

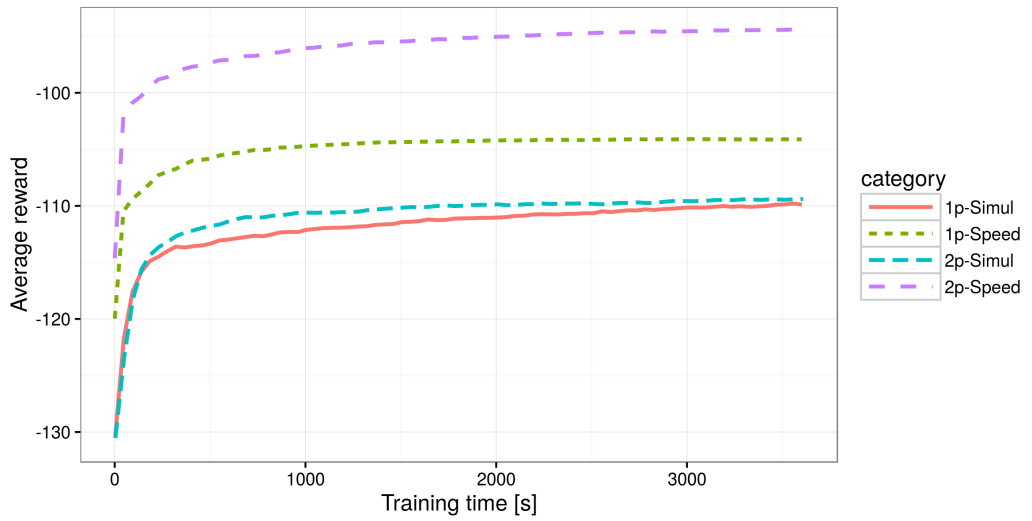


Figure 4.2 – Evolution of scores during training for various surrogate models

robot problems. The main reason behind the lack of difference is that solving the problem with two robots is significantly harder. During refining mutation, PML has to optimize functions with more parameters and it has also access to more actions. Moreover, since only orthogonal splits are considered in the current version of PML, a large number of splits are required to produce a policy in which the closest player to the ball will perform the kick.

While *1PSpeed* tend to slightly overestimate the approach cost, *2PSpeed* underestimates it significantly. In this case, we can see that the rough approximation of the approach cost has a major impact on the prediction of the cost.

The evolution of scores obtained during the training is presented in Figure 4.2. We note that the major part of the improvement is obtained during the first 500 seconds for all the models. However, after 1 hour, the average reward is still slowly improving.

An example of rollout with one of the policies trained by PML is shown in figure 4.3. At step 0, the robots are positioned randomly on the field, at a distance up to 4 meters of the ball. Although p2 is closer to the ball than p1, p1 takes the lead and approach the ball to perform a kick while p2 is moving toward the expected position of the ball. At step 1, both players have reached their targets, p1 has kicked the ball and we can see that the shoot was more powerful than expected. The chosen kicker is p2 who is much closer to the new position of the ball than p1. He performs the kick and successfully avoid the goalkeeper, scoring a goal before p1 reaches its target position.

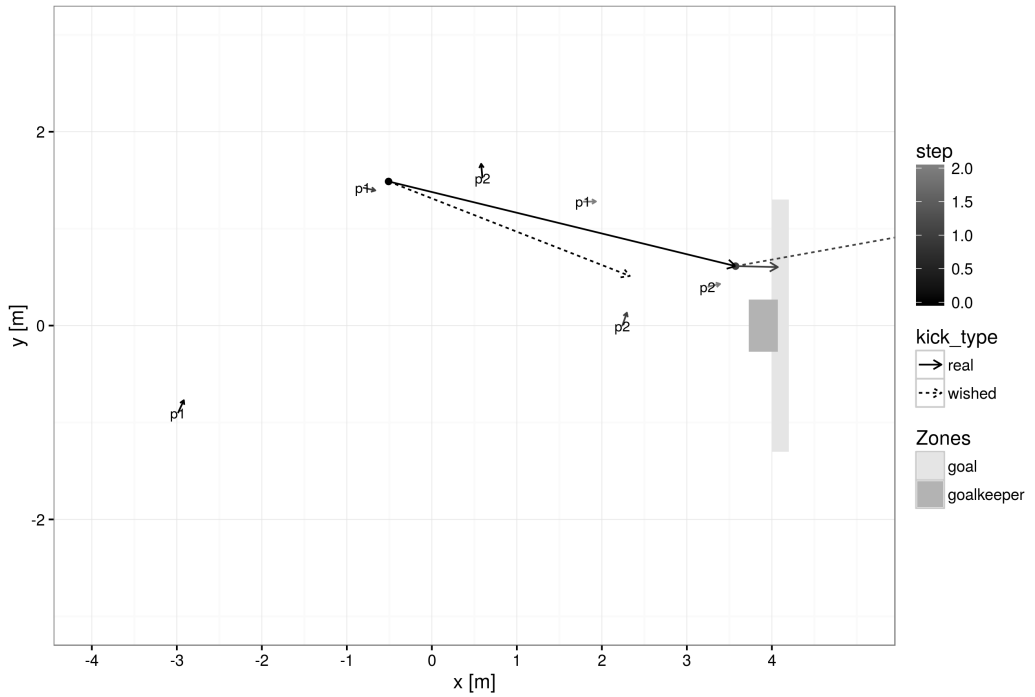


Figure 4.3 – Execution of a robot kicker problem after training with 2 robots

Table 4.2 – Impact of initial seed on the average cost after training

Problem	nb robots	Seed	Training without seed	Training with seed
Simul	1	132.75	113.21	110.43
	2	132.87	116.26	109.84
Speed	1	126.66	106.09	106.15
	2	126.67	98.03	94.04

4.2.2 Importance of initial policy

In order to measure the impact of using an expert seed on the training results, we present in table 4.2 the difference of expected cost after one hour of training depending on the use or not of an initial seed, see 4.1.1. In order to provide meaningful comparison, we also present the average cost received by the policy used as a seed for training.

All the policies were evaluated on the training problem, using 50 different training for each modality to produce policies and performing 1000 different rollouts per policy. The average performance of the seed policy on each problem was approximated using 50'000 rollouts. Due to the high dispersion of rewards received, even with 50'000 samples, there is still a small difference in the estimated cost for the expert policy between *1PSimul* and *2PSimul*. We can also observe that the average cost for the expert policy is significantly lower

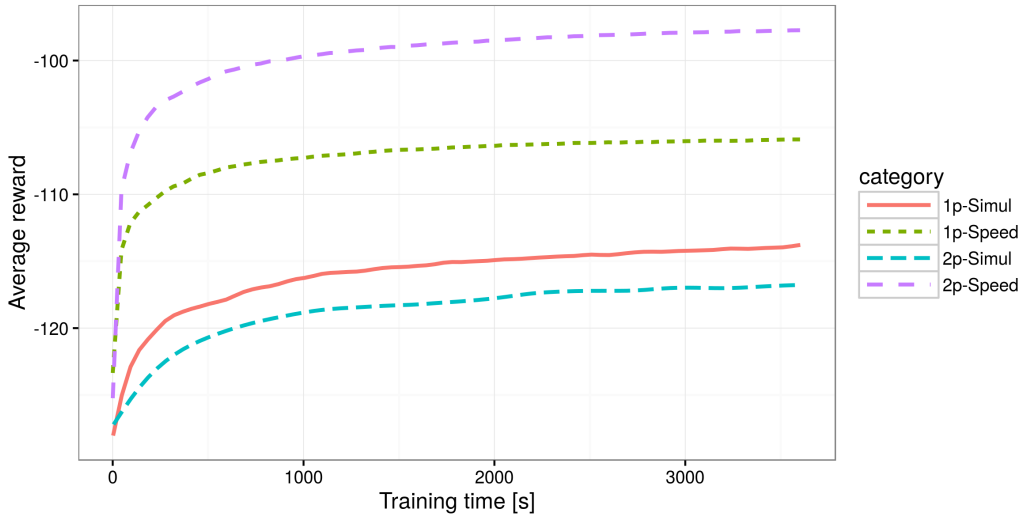


Figure 4.4 – Evolution of scores during training without initial seed

on *Speed* than on *Simul*. This indicates that the *Speed* problem underestimates the time needed for approach.

The general trend for ranking is as following: training with the expert seed provide better results than training without seed which was already better than the expert seed. There is a single exception for the *1PSpeed* problem, training without seed led to a slightly better reward than training with seed, but given the amount of data collected and the performance difference, the difference is not significant. From the entries presented in table 4.2, we can observe that the impact of the seed was particularly strong for *2PSimul*. This fact was expected since drawing samples from the *Simul* blackbox is more expensive, thus making the number of samples acquired during training lower.

The evolution of policy scores during the training process without initial seed is presented in figure 4.4. While the general trend is similar to the evolution of scores with an initial seed, see figure 4.2, the evolution tend to be slower, particularly during the the early phase. After one hour of training the average reward is still growing slowly indicating that additional time would still improve the average reward.

4.2.3 Combining offline learning and online planning

While we proposed only methods based on offline learning for the resolution of the *ball approach* problem in chapter 3, we experimented both type of methods for the *kicker robot* problem. There is two important differences between the two problems. First, the frequency at which actions need to be updated is much lower in the *kicker robot* problem. Second, the horizon of the problem is much shorter since the robots are generally able to score a goal in four steps,

Table 4.3 – Configurations of online planners for experiments

Problem	nb robots	EvalRollouts	MaxEvals	ValidationRollouts
Simul	1	20	30	200
	2	12	20	40
Speed	1	100	300	10000
	2	100	180	3000

Table 4.4 – Average cost received during offline and online evaluations

Problem	nb robots	Offline	Online	Offline+Online
Simul	1	110.43	108.28	103.49
	2	109.84	104.31	97.40
Speed	1	106.15	103.98	100.50
	2	94.04	94.08	90.78

even in the worst cases, while for *ball approach*, the robot can perform up to 100 steps before reaching a target position.

The online planner used in this experiment was described in section 2.5. In order to ensure that it could be used in embedded situations we tuned up the number of rollouts and evaluations to ensure that the average time spent for a rollout using the planner is approximately 1 second. This constraint led us to use the parameters shown in table 4.3. Due to the difference of time required for simulations, online planning performed using the *Simul* problem has access to a small number of rollouts.

In order to evaluate the contributions of the offline training and the online planning, we present the results of three different methods in table 4.4.

Offline Offline training is used with a budget of 1 hour on 20 threads. The results are directly imported from validation on *Simul* problem from section 4.2.1.

Online Online planning is used with the expert policy described in 4.1.1 as a rollback policy.

Offline+Online Offline training is used to produce a policy which is used as a rollback policy by the online planner.

Performances were evaluated using 50 different offline training and evaluating each policy with 1000 rollouts on the *Simul* problem, note that the problem used for online planning can be different from the problem used for evaluation. Purely online policies based on the expert policy were evaluated using 20'000 rollouts.

According to the results shown in table 4.4, using online planning on this problem tend to strongly outperform offline training on two player problems,

except for *Speed*. According to our understanding, this is because the offline training manages to distribute the role of kicking to the appropriate robot more easily with the *Speed* problem since it has access to more simulations. The combination of both methods produces significantly better results than each method taken individually. We can note that even for the *Simul* problem where the number of rollouts allowed is particularly low, results are still improved by online planning.

The best performance reached by offline training (94.04) is similar to the best performance reached by online planning(94.08). This result is particularly satisfying given the fact that a rollout using the policy trained offline costs less than 1[ms], 1000 times less than the budget allowed for rollouts based on online planning.

4.2.4 Evaluation of improvement through lateral kick

Since adjusting the final position of the ball require a significant amount of time, being able to kick the ball laterally seems a major advantage because it increases the options available to the kicker. We experimented the impact of introducing lateral kicks as available options and the empirical results are presented in table 4.5. Those results were obtained by training and planning on the *Simul* and *Speed* problems, while evaluation was always performed on *Simul* problem.

Table 4.5 – Impact of lateral kicks on average cost

Problem	nb robots	Offline		Offline+Online	
		classic	lateral kicks	classic	lateral kicks
Simul	1	110.43	109.17	103.49	102.44
	2	109.84	107.38	97.40	102.60
Speed	1	106.15	104.64	100.50	100.97
	2	94.04	102.96	90.78	91.24

Contrary to primary belief, adding lateral kicks do not significantly reduces the cost of scoring a goal, it even increases it in some cases. Our understanding of this fact is rather simple. Lateral kicks travel a shorter distance than powerful kicks and they present a more important noise on direction. Therefore, their use is limited to specific situations where the gain on approach is higher than the penalty of risking to have to perform one additional kick. While their use is limited to specific situations, they still consume a significant amount of the budget allocated for both online planning and offline training.

These results indicates that given the current specifications of the problem, developing and tuning up a lateral kick might not be worth, since this feature does not provide a clear benefit for scoring goals faster. Further investigation on this matter should include tests with modifications on the description of

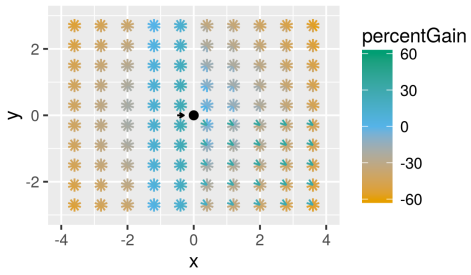
the lateral kick to check if improvements on repeatability or power could have a positive impact. Moreover, while lateral kick is not useful for reducing the time needed to score a goal, it might still be useful for avoiding opponents.

4.2.5 Cooperation

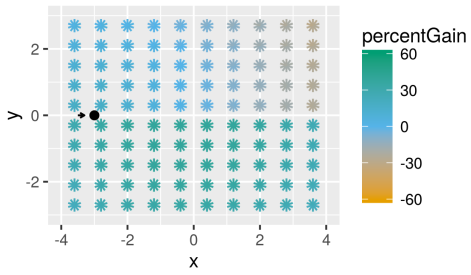
In order to highlight the differences between performances with 1 and 2 robot, we took out the best policies for 1 and 2 robots on the *Speed* problem. To avoid giving a major advantage to the 2 robots problem, we started with one of the robot positioned at 50 centimeters from the ball in both problem. For the two robots problem, we tested 1'000 different states, dividing each dimension into 10 cells. The average reward for each state was computed using 10'000 rollouts.

Consider an average cost for 1 robot problem $c_1 \in \mathbb{R}$ and an average cost of $c_2 \in \mathbb{R}$ for the 2 robot problem. We define the relative gain of playing in cooperation by: $\frac{c_1 - c_2}{c_1}$. Note that a negative relative gain implies that the cooperation led to an increase of the cost.

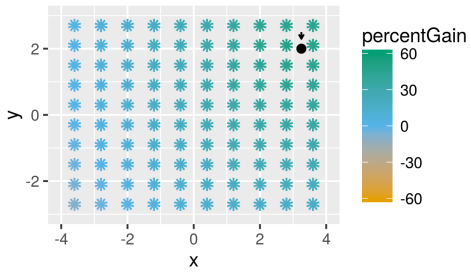
We show the difference of performances for three different initial positions of the ball and the first robot in Fig. 4.5. The initial position of the ball is represented by a black circle, the state of the first robot is represented by a black arrow and the relative gain is represented using a color scale.



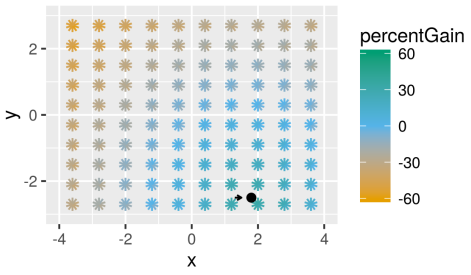
(a) b $= (0, 0)$, p1 $= (-0.5, 0, 0)$



(b) b $= (-3, 0)$, p1 $= (-3.5, 0, 0)$



(c) b $= (3.25, 2)$, p1 $= (3.25, 2.5, -\pi/2)$



(d) b $= (1.8, -2.5)$, p1 $= (1.3, -2.5, 0)$

Figure 4.5 – Representation of the gain for cooperation

In some situations, playing cooperatively according to the trained policies can reduce the cost by up to 60 %, while in others, it increases the cost by 60 %. Since there always exists 2 robots policy equivalent to the 1 robot policy, this proves that after 1 hour of computation, the 2 robots policy is suboptimal. This problem is mainly due to the difficulty of representing the natural choice of kicking with the closest robot with univariate splits based on the problem dimensions. We noticed two major discontinuities of gain due to univariate splits. First, based on p2 orientation in Fig. 4.5a. Second, based on p2 lateral position in Fig. 4.5b.

Despite the issue of univariate split, we saw in Table 4.1 that the 2 robots policy yield a better average reward than the 1 robot policy. This highlights that by supporting other types of splits, PML could provide a strong incentive for cooperation between robots.

4.3 Discussion

In this chapter, the kicker robot problem was presented along with the issues involved in team-play for robots. Experimental results led to various observations which can be used for this specific problem but also adapted to other problems.

Our experiments showed how using approximations of a problem can help to find more satisfying policies in a given amount of time by speeding up the simulation process. By presenting performances of PML during training with and without expert information provided as a seed, we showed how it is possible to include external knowledge about the problem although it is not mandatory to obtain satisfying results. We compared results obtained by online planners and offline training and showed how we can use a simple scheme to combine both strengths. Finally, a brief study of the impact of lateral kick was introduced, showing how model-based learning can be used to evaluate the impact of features in simulation, thus helping to guide the development of the robots.

While the modeling of the problem includes several key aspects of the real-world problem, only real-world experiments can confirm the applicability of our method to humanoid robot soccer.

Empirical results showed that online planning can strongly improve the performance of the decision making while using a moderate amount of computational power. By optimizing only the next step while relying on policies computed offline for the following steps, the burden on the online planner is reduced while still receiving a major improvement for problems with a limited horizon such as the kicker robot. The scheme proposed in this chapter could easily be improved by gathering data during offline training and then using them to guide the online sampling, allowing a smarter exploration for the

planner. Furthermore, using a smarter exploration could allow to use open-loop planning for a few steps which could represent an advantage on some problems.

By combining online planning and offline training, we hope to be able to tackle even more complex problems. Among the objectives we follow, we imagine to optimize policies offline on simplification of the problem and to refine them online using additional information. Among the targeted problems, policies could be trained for cooperation between robots and then used online by adding information about the position of the opponents.

Conclusion

In this thesis, three new algorithms for autonomous decision making of robots are presented: fitted policy forests (FPF), random forests policy iteration (RFPI) and policy mutation learner (PML). They focus on learning policies which can be used in real-time in order to provide to robots an efficient decision-making, thus ensuring their applicability in real-time applications. This thesis analyses in-depth two decision-making problems appearing in the RoboCup competition. Controlling the walk engine to approach the ball and deciding which robot has to kick the ball and how.

The first algorithm we designed was *fitted policy forests*. While it provides satisfying results on benchmark problems, its low sample efficiency makes it impractical for robotic applications where acquiring samples has a high cost, for example when it requires human supervision or involves risk of damaging the hardware.

In order to be able to tackle complex robotic problems, we changed the learning paradigm, by modeling the problems as blackbox functions with low computational cost. In order to reduce the discrepancy between the model and the real world, we decided to optimize coefficients of the blackbox function to match observations acquired from the real world. One of the advantages of this method is that a small number of samples can already reduce the discrepancy between the model and the real world.

The second algorithm we introduce is *random forests policy iteration*. It was experimented on the *ball approach problem*. Our experimental results show that it outperforms both the expert policy we used to win RoboCup 2016 and a version of this algorithm with coefficients optimized by CMA-ES. This is very satisfying. However, this solver is based on homogeneous action spaces and is therefore unsuited for problems with heterogeneous action spaces such as the *kicker robot problem*.

The last algorithm we propose is *policy mutation learner* (PML). Policies are represented as regression trees. Once computed offline, these policies can be used online to perform decision-making in real-time at a low computational cost. This algorithm performs successive local mutations of the regression tree representing the policy, seeking for mutations improving the current policy. PML can be seeded by an expert policy, and improve it further. PML is able to solve problems with heterogeneous action spaces, in particular the *kicker robot*

problem. While PML already provides satisfying results on the kicker robot problem, we point out multiple modifications which can lead to performance improvements in section 2.4.9.

Finally, we experimented the use of simple online planning using an offline policy as rollback. This scheme proved to be effective on the *kicker robot problem*, and can be used for decision-making problems with a large time step.

While the results obtained on the *ball approach problem* with RFPI are satisfying in both simulation and real-world experiments, there are still important improvements which need to be included. During RoboCup 2017, our team started to perform passes, thanks to improvement in localization accuracy and robustness. This revealed that taking into account an estimation of the ball speed in the state space would lead to a substantial improvement of the performances of the robot.

Experiments on the *kicker robot problem* showed that PML can be used for both improving expert policies or developing policies from scratch. While the results were satisfying, the policies produced can still be strongly improved, especially regarding the choice of the robot performing the kick as discussed in section 4.2.5. Real-world experimentation would also be needed to ensure that the training in simulation has a positive impact in real-world situations. Although our learning experiments on the *ball approach problem* strongly outperformed those obtained by expert policies, our procedure was not mature enough to be used during RoboCup 2017 but we plan to use it for RoboCup 2018.

Applications of our decision-making algorithms are based on modeling the problem with a blackbox based on real-world samples and then optimize policies based on the blackbox which is an approximation of the real-world. Among the consequences, benefits of using the policy tend to be greater in simulation than in real-world. According to the theory, the policy produced by RFPI was supposed to take 2.7 times less steps than the one used in RoboCup 2016. In practice it was 1.7 times faster. While it is still an improvement, it appears that reducing the discrepancy between the blackbox model and the reality could strongly help to improve the performances of the trained policies in the real world. One of the problems we identified in our calibration of the predictive motion model is the fact that the noise applied at every step is based on a rough expert estimation and not on real data. By using approaches based on likelihood maximization [Roy and Thrun 1999] to calibrate the model, we might be able to improve our results.

Learning accurate blackbox models is as important as finding efficient policies in robotics. In order to make the best decisions, robots need to have a model of the results of their actions. While this model might be provided by humans and based on data acquired in the real world, it is often a time-consuming task. In order to face this issue, we think that we should aim towards robots autonomously learning the results of their actions rather than relying on exter-

nal measurements. At RoboCup, this would allow teams to simply place their robots on the field and let them optimize their models autonomously. From a more general point of view, autonomously learning approximated blackbox models corresponding to situations they face would allow robots to constantly improve and adapt to new problems.

Appendix A

Blackbox optimizers

This chapter presents four methods used to optimize blackbox function. A brief explanation of each algorithm is provided along with a description of the process of tuning parameters and additional references for further reading.

A.1 CMA-ES

Covariance matrix adaptation evolution strategy, CMA-ES for short, is a blackbox optimizer proposed in [Hansen and Ostermeier 2001]. This algorithm uses a limited memory by tracking and updating the mean and the covariance matrix of a multivariate normal distribution. It is essentially a local optimizer which is strongly influenced by the initial guess and the initial step size.

At each step, a new generation of inputs are samples from the multivariate distribution, the mean of the distribution is updated by using a weighted average of the best inputs, where the input with higher rewards have higher weights. The evolution of the covariance matrix depends on the weighted difference of best inputs toward their weighted average. Note that evolution of the covariance matrix uses a changing rate to ensure some continuity in the dispersion.

CMA-ES possesses many parameters such as the size of the population at every step, the number of samples to consider for the weighted mean and the initial guess. However, several heuristics have been developed to choose them automatically depending on the input space, thus allowing to obtain satisfying results when using the algorithm out-of-the-box, even on high-dimensional spaces.

For the scope of this thesis, we use libcmaes[1] a C++ implementation of CMA-ES developed by Emmanuel Benazera. CMA-ES was designed to optimize deterministic blackbox functions but libcmaes implements a simple scheme to deal with stochastic blackbox functions: it simply multiplies the

[1] https://github.com/beniz/libcmaes

number of samples drawn at each step by 5. However, our experiments highlighted that the number 5 might be highly insufficient for optimizing functions with a large amount of noise with respect to local variations of the mean reward. Uncertainty handling for CMA-ES, UH-CMA-ES for short, is presented in [Heidrich-Meisner and Igel 2009], this method control the signal to noise ratio and automatically adapts the number of evaluations to the blackbox function at each generation.

A.2 Cross entropy

The cross entropy algorithm, CE for short, is a limited memory blackbox optimizer introduced in [Rubinstein 1999]. It keeps in memory the parameters of a distribution over the input space. And update the candidate by moving toward areas of the space with the highest rewards. It performs a local search which is strongly influenced by the initial distribution provided.

At every step, it samples n inputs from the current distribution and evaluates their values using the blackbox. The k more promising inputs are selected and used to fit a new distribution which is used as a basis for the next step. The end of the algorithm is obtained either when the parameters of the distribution have converged or after a number of steps defined by the user.

The four parameters of the algorithms are: the number of inputs sampled at each generation, the number of inputs conserved for estimating the new distribution, the number of generations of the process and finally the shape of the distribution which is usually a multivariate normal distribution. Those parameters are critical for the performance of the CE algorithm. Note that Fully automated cross entropy, FACE for short, is a variation of CE in which the size of the population changes dynamically during execution. This algorithm was proposed in [De Boer *and al.* 2005] with an intention of identifying problems where CE would not be able to provide a reliable solution.

A.3 Simulated annealing

The simulated annealing algorithm, SA for short, is a limited memory global optimizer inspired by thermodynamic principles since it only stores one input and the index of the current iteration. It has been used to optimize very large problem such the traveling salesman problem with several thousands of cities in [Kirkpatrick *and al.* 1983], therefore it is not limited to optimizing blackboxes with continuous input. While simulated annealing perform a search in the global state space, it is not sample efficient and is therefore more suited for cheap blackbox functions.

At every step of the algorithm, a neighbor is sampled by applying modifications to the current candidate. The value of the new candidate is computed

and then, the new input is always selected if its value is higher than the value of the current candidate. There is still a probability of accepting less performing state which depends on the current temperature of the optimizer and the loss resulting of accepting the new input. Typically, the Boltzmann distribution is used and acceptance probability $p(r, r', T) = e^{-\frac{r-r'}{T}}$, with r the reward of the current candidate, r' the reward of the new input and T the current temperature of the process. The temperature generally decreases with time, thus strongly reducing the probability of accepting new candidates which would decrease the reward. There are different cooling schemes for the temperature, some using plateau, others chaining multiple phases of warming and cooling.

The two parameters of the simulated annealing optimizer are the number of samples allowed and the temperature evolution. High temperature is associated with global search since it allows to take local steps which are reducing the reward, but which might have neighbors yielding higher rewards. This impact can also be increased by using temperature to influence the neighborhood function. For continuous problems, the standard deviation of the distribution from which neighbors are sampled might be proportional to the temperature.

Originally designed to solve deterministic problems, SA requires to sample several times stochastic blackbox function for each input in order to yield satisfying results. If the blackbox function is sampled only once, then the algorithm tend to converge to an input which might have a low average reward, but chances to produce a high reward.

In this thesis, we only consider stochastic blackbox for continuous problems, see **??**. We use a simple linear profile for temperature, see Eq. (A.1), with k the current iteration, n the number of iterations allowed and T_0 the initial temperature.

$$T(k, n) = T_0 \frac{n - k}{n} \tag{A.1}$$

Since this thesis focuses on continuous problems with a bounded input space, we decided to use a simple scheme for the exploration: at each step we sample the next element according to Eq. (A.2). With i_{k+1} the new input to sample at iteration $k+1$, c_k the candidate in memory after iteration k, $\mathcal{A}(I)i$ the amplitude of space I along dimension i and d the number of dimensions of the input space.

$$i_{k+1} = c_k + \mathcal{U}\left(\begin{bmatrix} -\mathcal{A}(I)1 & \mathcal{A}(I)1 \\ \dots & \dots \\ -\mathcal{A}(I)d & \mathcal{A}(I)d \end{bmatrix}\right) \frac{T(k, n)}{T_0} \tag{A.2}$$

A.4 Bayesian optimization

Bayesian optimization is a state-of-the-art method for global optimization. Its main principle is to use the samples acquired by interacting with the black-box function to establish a predictive model for the function based on Gaussian processes. Since Gaussian processes provide confidence bounds for different inputs, it is then possible to use an acquisition function which is based on both: the estimated mean and the uncertainty. While several acquisition functions have been proposed, an interesting approach has been proposed in [Hoffman *and al.* 2011]: it consists of using portfolio containing several acquisition functions and sample from them using a bandit-based approach.

While Bayesian optimization methods provides unprecedented sample efficiency, they come with high cost for the acquisition function and are therefore suited for optimizing expensive blackboxes. For cheap blackbox functions in high-dimensional spaces, it is generally more computationally efficient to acquire more samples, even if the choice of the samples is clearly sub-optimal.

Bibliography

Atkeson, Christopher G., Moorey, Andrew W., Schaalz, Stefan, Moore, Andrew W and Schaal, Stefan, 1997. Locally Weighted Learning. *Artificial Intelligence*, 11:11–73. doi:10.1023/A:1006559212014.

Behnke, Sven, 2006. Online trajectory generation for omnidirectional biped walking. *Proceedings - IEEE International Conference on Robotics and Automation*, 2006(May):1597–1603. doi:10.1109/ROBOT.2006.1641935.

Benbrahim, H, Doleac, J S, Franklin, J A and Selfridge, O G, 1992. Real-time learning: a ball on a beam. *International Joint Conference on Neural Networks, 1992. IJCNN*, 1:98–103. doi:10.1109/IJCNN.1992.287219.

Bernstein, Andrey and Shimkin, Nahum, 2010. Adaptive-resolution reinforcement learning with polynomial exploration in deterministic domains. *Machine Learning*, 81(1999):359–397. doi:10.1007/s10994-010-5186-7.

Bertsekas, Dimitri P., 2013. Lambda-Policy Iteration: A Review and a New Implementation. In *Reinforcement Learning and Approximate Dynamic Programming for Feedback Control*, pages 379–409. ISBN 9781118104200. doi:10.1002/9781118453988.ch17.

Borchani, Hanen, Varando, Gherardo, Bielza, Concha and Larrañaga, Pedro, 2015. A survey on multi-output regression. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 5(5):216–233. doi:10.1002/widm.1157.

Breiman, L, Friedman, J H, Olshen, R A and Stone, C J, 1984. *Classification and Regression Trees*, volume 19. ISBN 0412048418. doi:10.1371/journal.pone.0015807.

Breiman, Leo, 1996. Bagging predictors. *Machine Learning*, 24(2):123–140. doi:10.1007/BF00058655.

Brochu, Eric, Cora, Vlad M and December, Freitas, 2010. A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning.

De Boer, Pieter Tjerk, Kroese, Dirk P., Mannor, Shie and Rubinstein, Reuven Y., 2005. A tutorial on the cross-entropy method. *Annals of Operations Research*, 134(1):19–67. doi:10.1007/s10479-005-5724-z.

Deisenroth, Marc P and Rasmussen, Carl Edward, 2011. PILCO: A Model-Based and Data-Efficient Approach to Policy Search. *Icml*, pages 465–472.

Ernst, Damien, Geurts, Pierre and Wehenkel, Louis, 2005. Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research*, 6(1):503–556.

Geurts, Pierre, Ernst, Damien and Wehenkel, Louis, 2006. Extremely randomized trees. *Machine Learning*, 63(1):3–42. doi:10.1007/s10994-006-6226-1.

Gittins, Jc, Gittins, Jc, Jones, Dm and Jones, Dm, 1979. A dynamic allocation index for the discounted multiarmed bandit problem. *Biometrika*, 66(3):561–565. doi:10.2307/2335176.

Hansen, Nikolaus, Müller, Sibylle D and Koumoutsakos, Petros, 2003. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). *Evolutionary computation*, 11(1):1–18. doi:10.1162/106365603321828970.

Hansen, Nikolaus and Ostermeier, Andreas, 2001. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation*, 9(2):159–195. doi:10.1162/106365601750190398.

Heidrich-Meisner, Verena and Igel, Christian, 2009. Uncertainty handling CMA-ES for reinforcement learning. *Proceedings of the 11th Annual conference on Genetic and evolutionary computation - GECCO '09*, page 1211. doi:10.1145/1569901.1570064.

Hofer, Ludovic and Gimbert, Hugo, 2016. Online Reinforcement Learning for Real-Time Exploration in Continuous State and Action Markov Decision Processes. In *PlanRob2016, Proceedings of the 4th Workshop on Planning and Robotics at ICAPS2016*. AAAI.

Hofer, Ludovic and Rouxel, Quentin, 2017. An Operational Method Toward Efficient Walk Control Policies for Humanoid Robots. In *ICAPS 2017*.

Hoffman, Matthew, Brochu, Eric and Freitas, Nando De, 2011. Portfolio Allocation for Bayesian Optimization. *Conference on Uncertainty in Artificial Intelligence*, pages 327–336.

Ijspeert, Auke Jan, Nakanishi, Jun, Hoffmann, Heiko, Pastor, Peter and Schaal, Stefan, 2013. Dynamical movement primitives: learning attractor models for motor behaviors. *Neural computation*, 25(2):328–73. doi: 10.1162/NECO_a_00393.

Kirkpatrick, S., Gelatt, C. D. and Vecchi, M. P., 1983. Optimization by Simulated Annealing. *Science*, 220(4598):671–680. doi:10.1126/science.220.4598.671.

Kober, Jens, Kober, Jens and Peters, Jan, 2009. Policy search for motor primitives in robotics. *NIPS 2008*, pages 849—-856.

Kober, Jens and Peters, Jan, 2012. Reinforcement Learning in Robotics: A Survey. In Marco Wiering and Martijn van Otterlo, editors, *Reinforcement Learning: State-of-the-Art*, pages 579–610. Springer Berlin Heidelberg.

Koenig, Sven and Simmons, Reid G., 1996. The effect of representation and knowledge on goal-directed exploration with reinforcement-learning algorithms. *Machine Learning*, 22(1-3):227–250. doi:10.1007/BF00114729.

Kohavi, Ron, 1995. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *Appears in the International Joint Conference on Articial Intelligence (IJCAI)*, pages 1–7. ISBN 1-55860-363-8. doi:10.1067/mod.2000.109031.

Lemon, Stephenie C, Roy, Jason, Clark, Melissa A, Friedmann, Peter D and Rakowski, William, 2003. Classification and Regression Tree Analysis in Public Health: Methodological Review and Comparison With Logistic Regression. *The Society of Behavioral Medicine*, 26(3):172–181. doi:Doi10.1207/S15324796abm2603_02.

Li, Lihong, Littman, Michael L. and Mansley, Christopher R., 2009. Online exploration in least-squares policy iteration. *The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 733–739.

Loh, Wei-Yin, 2011. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):14–23. doi: 10.1002/widm.8.

McCallum, A K, 1996. *Reinforcement learning with selective perception and hidden state*. PhD thesis.

McGeer, T., 1990. Passive Dynamic Walking. *The International Journal of Robotics Research*, 9(2):62–82. doi:10.1177/027836499000900206.

Meuleau, Nicolas, Benazera, Emmanuel, Brafman, Ronen I., Hansen, Eric A. and Mausam, 2009. A Heuristic Search Approach to Planning. *J. Artif. Int. Res.*, 34:27–59.

Nemec, Bojan, Zorko, Matej and Zlajpah, Leon, 2010. Learning of a ball-in-a-cup playing robot. In *19th International Workshop on Robotics in Alpe-Adria-Danube Region, RAAD 2010 - Proceedings*, pages 297–301. ISBN 9781424468867. doi:10.1109/RAAD.2010.5524570.

Ng, Andrew Y and Jordan, Michael, 2000. PEGASUS: A Policy Search Method for Large MDPs and POMDPs. *Conference on Uncertainty in Artificial Intelligence*, 94720:406–415. doi:10.1.1.81.1085.

Ng, Andrew Y, Kim, H Jin, Jordan, Michael I and Sastry, Shankar, 2004. Inverted autonomous helicopter flight via reinforcement learning. In *International Symposium on Experimental Robotics*.

Nouri, Ali and Littman, Michael L., 2009. Multi-resolution Exploration in Continuous Spaces. *Advances in Neural Information Processing Systems*, pages 1209–1216.

Pazis, Jason and Lagoudakis, Michail G, 2009. Binary action search for learning continuous-action control policies. *Proceedings of the 26th International Conference on Machine Learning (ICML)*, pages 793–800. doi: http://doi.acm.org/10.1145/1553374.1553476.

Peters, Jan and Schaal, Stefan, 2006. Policy Gradient Methods for Robotics. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2219–2225. ISBN 1-4244-0258-1. doi:10.1109/IROS.2006.282564.

Rasmussen, Carl Edward, 2006. Gaussian processes for machine learning. *International journal of neural systems*, 14(2):69–106. doi:10.1142/S0129065704001899.

Rasmussen, Carl Edward and Deisenroth, Marc Peter, 2008. Probabilistic Inference for Fast Learning in Control. *Learning*, (July).

Rouxel, Quentin, Passault, Gregoire, Hofer, Ludovic, N'Guyen, Steve and Ly, Olivier, 2016. Learning the odometry on a small humanoid robot. In Danica Kragic, Antonio Bicchi and Alessandro De Luca, editors, *2016 {IEEE} International Conference on Robotics and Automation, {ICRA} 2016, Stockholm, Sweden, May 16-21, 2016*, pages 1810–1816. IEEE. ISBN 978-1-4673-8026-3. doi:10.1109/ICRA.2016.7487326.

Roy, Nicholas and Thrun, Sebastian, 1999. Online self-calibration for mobile robots. In *Robotics and {Automation}, 1999. {Proceedings}. 1999 {IEEE} {International} {Conference} on*, volume 3, pages 2292–2297. IEEE.

Rubinstein, Reuven Y, 1999. The Cross-Entropy Method for Combinatorial and Continuous Optimization. *Methodology and Computing in Applied Probability*, 1(2):127–190. doi:10.1007/s11009-008-9101-7.

Sanner, Scott, Delgado, Karina Valdivia and de Barros, Leliane Nunes, 2012. Symbolic Dynamic Programming for Discrete and Continuous State MDPs. In *Proceedings of the 26th Conference on Artificial Intelligence*, volume 2. ISBN 978-0-9749039-7-2.

Santamaria, J. C., Sutton, R. S. and Ram, A., 1997. Experiments with Reinforcement Learning in Problems with Continuous State and Action Spaces. *Adaptive Behavior*, 6(2):163–217. doi:10.1177/105971239700600201.

Scherrer, Bruno, 2013. Performance bounds for λ policy iteration and application to the game of tetris. *Journal of Machine Learning Research*, 14(1):1181–1227.

Silver, David, Huang, Aja, Maddison, Chris J., Guez, Arthur, Sifre, Laurent, van den Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, Dieleman, Sander, Grewe, Dominik, Nham, John, Kalchbrenner, Nal, Sutskever, Ilya, Lillicrap, Timothy, Leach, Madeleine, Kavukcuoglu, Koray, Graepel, Thore and Hassabis, Demis, 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489. doi:10.1038/nature16961.

Wang, H O, Tanaka, K and Griffin, M F, 1996. An approach to fuzzy control of nonlinear systems: Stability and design issues. *Ieee Transactions on Fuzzy Systems*, 4(1):14–23. doi:10.1109/91.481841.

Weinstein, Ari and Littman, ML, 2013. Open-Loop Planning in Large-Scale Stochastic Domains. In *27th AAAI Conference on Artificial Intelligence*, volume 1, pages 1436–1442. ISBN 9781577356158.

Williams, Ronald and Baird, Leemon C., 1994. Tight Performance Bounds on Greedy Policies Based on Imperfect Value Functions. *Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems*, pages 108–113.