

Online Reinforcement Learning for Real-Time Exploration in Continuous State and Action Markov Decision Processes

Hofer Ludovic and Gimbert Hugo

Laboratoire Bordelais de Recherche en Informatique
351 Cours de Libération
33405 Talence
France

Abstract

This paper presents a new method to learn online policies in continuous state, continuous action, model-free Markov decision processes, with two properties that are crucial for practical applications. First, the policies are implementable with a very low computational cost: once the policy is computed, the action corresponding to a given state is obtained in logarithmic time with respect to the number of samples used. Second, our method is versatile: it does not rely on any a priori knowledge of the structure of optimal policies. We build upon the Fitted Q-iteration algorithm which represents the Q-value as the average of several regression trees. Our algorithm, the Fitted Policy Forest algorithm (FPF), computes a regression forest representing the Q-value and transforms it into a single tree representing the policy, while keeping control on the size of the policy using resampling and leaf merging. We introduce an adaptation of Multi-Resolution Exploration (MRE) which is particularly suited to FPF. We assess the performance of FPF on three classical benchmarks for reinforcement learning: the "Inverted Pendulum", the "Double Integrator" and "Car on the Hill" and show that FPF equals or outperforms other algorithms, although these algorithms rely on the use of particular representations of the policies, especially chosen in order to fit each of the three problems. Finally, we exhibit that the combination of FPF and MRE allows to find nearly optimal solutions in problems where ϵ -greedy approaches would fail.

1 Introduction

The initial motivation for the research presented in this paper is the optimization of closed-loop control of humanoid robots, autonomously playing soccer at the annual Robocup competition¹. We specifically target to learn behaviors on the Grosban robot, presented in Figure 1. This requires the computation of policies in Markov decision processes where 1) the state space is continuous, 2) the action space is continuous, 3) the transition function is not known. Additionally, in order to provide real-time closed-loop control, the policy should allow to retrieve a nearly optimal-action at a low computational-cost. We consider that the transition function is not known, because with small and low-cost humanoid

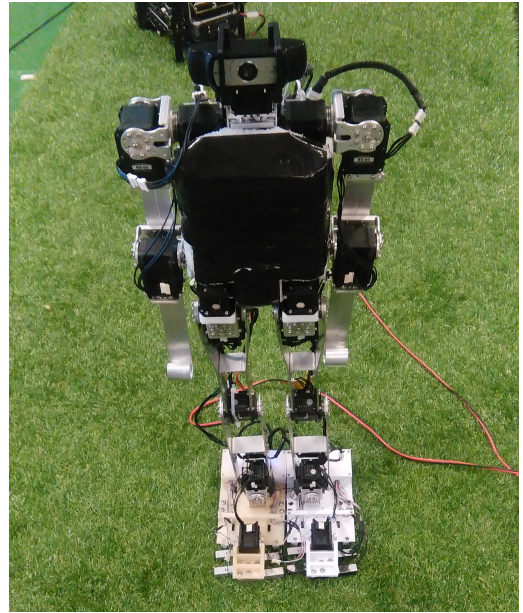


Figure 1: The Grosban robot

robots, the lack of accuracy on sensors and effectors makes the system behavior difficult to predict.

More generally, the control of physical systems naturally leads to models with continuous-action spaces, since one typically controls the position and acceleration of an object or the torque sent to a joint. While policy gradients methods have been used successfully to learn highly dynamical tasks such as hitting a baseball with an anthropomorphic arm (Peters and Schaal 2008), those algorithms are not suited for learning on low-cost robots, because they need to provide a motor primitive and to be able to estimate a gradient of the reward with respect to the motor primitive parameters. While model-based control is difficult to apply on such robots, hand-tuned open-loop behaviors have proven to be very effective (Behnke 2006). Therefore, model-free learning for CSA-MDP appears as a promising approach to learn such behaviors.

Since the transition and the reward functions are not known *a priori*, sampling is necessary. While an efficient ex-

¹http://wiki.robocup.org/wiki/Humanoid_League

exploitation of the collected samples is required, it is not sufficient. A smart exploration is necessary, because on some problems, nearly-optimal strategies requires a succession of actions which is very unlikely to occur when using uniform random actions. On extreme cases, it might even lead to situation where no reward is ever seen, because the probability of reaching a state carrying a reward while following a random policy is almost 0. This problem is known as the combinatorial lock problem and appears in discrete case in (Koenig and Simmons 1996) and in continuous problems in (Li, Littman, and Mansley 2009).

For control problems where the action set is discrete and not too large, there are already existing efficient algorithms to tackle the problem of producing an efficient policy from the result of previous experiments. Of course, these algorithms can be used in the continuous action space case, by discretization of the action sets. However this naive approach often leads to computational costs that are too high for practical applications, as stated in (Weinstein 2014).

The specificity of continuous action space has also been addressed with specific methods and particularly encouraging empirical results have been obtained thanks for example to the *Binary Action Search* approach (Pazis and Lagoudakis 2009), see also (Busoniu et al. 2010). These methods require to design functional basis used to represent the Q -value function, which we prefer to avoid in order to obtain versatile algorithms.

A recent major-breakthrough in the field of solving CSA-MDP is Symbolic Dynamic Programming which allows to find exact solutions by using eXtended Algebraic Decision Diagrams (Sanner, Delgado, and de Barros 2012), see also (Zamani, Sanner, and Fang 2012). However, those algorithms requires a model of the MDP and rely on several assumptions concerning the shape of the transition function and the reward function. Additionally, those methods are suited for a very close horizon and are therefore not suited for our application.

While local planning allows to achieve outstanding control on high-dimensionnal problems such as humanoid locomotion (Weinstein and Littman 2013), the computational cost of online planning is a burden for real-time application. This is particularly relevant in robotics, where processing units have to be light and small in order to be embedded. Therefore, we aim at global planning, where the policy is computed offline and then loaded on the robot.

Our own learning algorithms are based on the Fitted Q Iteration algorithm (Ernst, Geurts, and Wehenkel 2005) which represents the Q -value as the average of several regression trees. We first present a method allowing to extract approximately optimal continuous action from a Q -value forest. Then we introduce a new algorithm, Fitted Policy Forest (FPF), which learn an approximation of the policy function using regression forests. Such a representation of the policy allows to retrieve a nearly optimal action at a very low computational cost, therefore allowing to use it on embedded systems.

We use an exploration algorithm based on MRE (Nouri and Littman 2009), an optimistic algorithm which represents the knownness of state and action couples using a kd-

tree (Preparata and Shamos 1985). Following the idea of extremely randomized trees (Geurts, Ernst, and Wehenkel 2006), we introduce randomness in the split, thus allowing to grow a forest in order to increase the smoothness of the knownness function. Moreover, by changing the update rule for the Q -value, we reduce the attracting power of local maxima.

The viability of FPF is demonstrated by a performance comparison with the results proposed in (Pazis and Lagoudakis 2009) on three classical benchmark in RL: *Inverted Pendulum Stabilization*, *Double Integrator* and *Car on the Hill*. Experimental results show that FPF drastically reduce the computation time while improving performance. We further illustrate the gain obtained by using our version of MRE on the *Inverted Pendulum Stabilization* problem, we finally present the results obtained on the *Inverted Pendulum Swing-Up*, using an underactuated angular joint. This last experiment is run using Gazebo simulator in place of the analytical model.

This paper is organized as follows: Section 2 introduces the notations used for Markov decision processes and regression forests, Section 3 presents the original version of *Fitted Q -Iteration* and other classical methods in batch mode RL with continuous action space, Section 4 proposes algorithms to extract informations from regression forest, Section 5 introduces the core of the FPF algorithm. Section 6 presents the exploration algorithm we used. The efficiency of FPF and MRE is demonstrated through a series of experiments on classical RL benchmarks in section 7, the meaning of the experimental results is discussed in Section 8.

2 Background

2.1 Markov-Decision Process

A *Markov-Decision Process*, or MDP for short, is a 5-tuple $\langle S, A, R, T, \gamma \rangle$, where S is a set of states, A is a set of actions, R is a reward function ($R(s, a)$ denotes the expected reward when taking action a in state s), T is the transition function ($T(s, a, s')$ denotes the probability of reaching s' from s using a) and $\gamma \in [0, 1]$ is a discount factor.

A *Deterministic Policy* is a mapping $\pi : S \mapsto A$, where $\pi(s)$ denotes the action choice in state s . Thereafter, by “policy”, we implicitly refer to deterministic policy. The Q -value of a couple (s, a) under a policy π with an horizon H is denoted $Q_H^\pi(s, a)$ and is defined as the expected cumulative and discounted reward by applying a in state s and then choosing actions according to π :

$$Q_H^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') Q_{H-1}^\pi(s', \pi(s'))$$

We further abbreviate Q_∞^π by Q^π for short. The greedy policy with respect to Q is denoted π_Q and always selects the action with the highest Q -value; i.e. $\pi_Q(s) = \underset{a \in A}{\operatorname{argmax}} Q(s, a)$.

Considering that the action space is bounded to an interval, such a limit exists, although it is not necessarily unique.

It is known that an optimal Q -value function exists (Puterman 1994): $Q^* = \max_{\pi} Q^\pi$. The optimal policy π^* is greedy with respect to Q^* : $\pi^* = \pi_{Q^*}$.

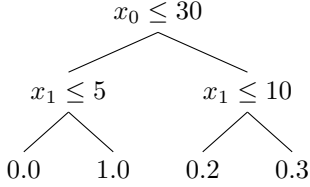


Figure 2: A simple regression tree

Given a complete and finite MDP, standard algorithms exist for finding the optimal policy, including value iteration, policy iteration and linear programming. However, if the transition function or the reward function are unknown, it is necessary to use samples to learn an approximation of the Q -value denoted \widehat{Q} . If the state space or the action space are continuous, it is also necessary to approximate the solution.

When solving offline a MDP while having no direct access to the transition function, it is necessary to use a set of gathered samples. Samples are defined as 4-tuples of the form: $\langle s, a, r, s' \rangle$ where s is the starting state, a the action used, r the reward received and s' the successor state.

2.2 Regression Forests

A *regression tree* is a representation of the approximation of a function $f : X \mapsto Y$ where $X \in \mathbb{R}^k$ and $Y \in \mathbb{R}$. It has a decision tree structure where every non-leaf node is a function mapping X to its children and every leaf is a basic function $\phi : X \mapsto Y$. A simple regression tree with piecewise constant (PWC) approximation is presented in Figure 2. Several algorithms exist to extract regression trees from training set, for a complete introduction, refer to (Loh 2011). Predicting the output y from an entry x requires to find the leaf corresponding to x and then to compute $\phi(x)$, with ϕ the basic function found at the leaf corresponding to x . We will further refer to the value predicted by a tree t for input x by $t(x)$ for short. While some algorithms use oblique split (Li, Lue, and Chen 2000), the algorithms presented here are only valid for orthogonal splits (splits of the form $x_i \leq v$). We will further note $LC(n)$ and $UC(n)$ the lower and upper children of node n , concerning $x_i \leq v$ and $x_i > v$ respectively.

If we define the space X as a hyperrectangle \mathcal{H} , each leaf will concern a different part of \mathcal{H} . We will further refer to the minimum and maximum value of \mathcal{H} along the dimension i as $\mathcal{H}_{i,m}$ and $\mathcal{H}_{i,M}$ respectively. We define the size of a hyperrectangle \mathcal{H} by $|\mathcal{H}| = \prod_{i=1}^{\dim X} \mathcal{H}_{i,M} - \mathcal{H}_{i,m}$. We use an abusive notation of the norm $\|\mathcal{H}\|$ in place of $\|\mathcal{H}_{i,M} - \mathcal{H}_{i,m}\|$.

A *regression forest* is a set of regression trees: $F = \{t_1, \dots, t_M\}$. It has been exhibited in (Breiman 1996) that using multiple trees to represent the function leads to a more accurate prediction. The value predicted by a forest F for an input x is $F(x) = \sum_{k=1}^M \frac{t_k(x)}{M}$.

2.3 Kd-trees

Kd-trees are a data structure which allows to store points of the same size while providing an $O(\log(n))$ access (Preparata and Shamos 1985). At each leaf of the tree, there is one or several points and at each non-leaf node, there is an orthogonal split. Let X be the space on which the kd-tree τ is defined, then for every $x \in X$, there exist a single path from the root of the kd-tree to the leaf in which x would fit. This leaf is denoted $\text{leaf}(\tau, x)$ is defined on the space X . Each leaf l contains a set of points noted $\text{points}(l)$ and concerns an hyperrectangle $\mathcal{H} = \text{space}(l)$.

3 Previous Work

The use of regression forests to approximate the Q -value of a continuous MDP has been introduced in (Ernst, Geurts, and Wehenkel 2005) under the name of *Fitted Q Iteration*. This algorithm uses an iterative procedure to build \widehat{Q}_H , an approximation of the Q -value function at horizon H . It builds a regression forest by using \widehat{Q}_{H-1} and a set of 4-tuples using the rules given at Equations 1 and 2.

$$x = (s, a) \quad (1)$$

$$y = r + \max_{a \in A} \widehat{Q}_{H-1}(s', a) \quad (2)$$

While this procedure yields very satisfying results when the action space is discrete, the computational complexity of the max part in equation 2 when using regression forest makes it become quickly inefficient. Therefore, in (Ernst, Geurts, and Wehenkel 2005), action spaces are always discretized to compute this equation, thus leading to an inappropriate action set when optimal control requires a fine discretization.

Binary Action Search, introduced in (Pazis and Lagoudakis 2009) proposes a generical approach allowing to avoid the computation of the max part in equation 2. Results presented in (Pazis and Lagoudakis 2009) show that Binary Action Search strongly outperforms method with a finite number of actions on two problems with rewards including a cost depending on the square of the action used: *Inverted Pendulum Stabilization* and *Double Integrator*. On the other hand, binary action search yields unsatisfying results on *Car on the Hill*, a problem with an optimal strategy known to be “bang-bang” (i.e. optimal strategy is only composed of two actions).

4 Approximation of the Q -value forest

In this part, we propose new methods to extract information from a regression forest while choosing a trade-off between accuracy and computational cost. First, we introduce the algorithm we use to grow regression forest. Then we present an algorithm to project a regression tree on a given subspace. Finally we propose a method allowing to average a whole regression forest by a single regression tree whose number of leaf is bounded.

4.1 Extra-Trees

While several methods exist to build regression forests from a training samples, our implementation is based on Extra-Trees (Geurts, Ernst, and Wehenkel 2006). This algorithm produces satisfying approximation at a moderate computational cost.

The main characteristic of Extra-trees is that k split dimensions are chosen randomly, then for each chosen split dimension the position of the split is picked randomly from an uniform distribution from the minimal to the maximal value of the dimension along the samples to split. Finally, only the best of the k random splits is used; the criteria used to rank the splits is the variance gain brought by the split. The original training set is splitted until one of the terminal condition is reached. The first terminal condition is that the number of samples remaining is smaller than n_{\min} , where n_{\min} is a parameter allowing to control overfitting. There are two other terminal conditions: if the inputs of the samples are all identical or if the output value is constant.

4.2 Improving Extra-trees

We provide two improvements to Extra-trees, in order to remedy two problems. First, due to the terminal conditions, large trees are grown for parts of the space where the Q -value is almost constant because if the Q -value is not strictly constant, the only terminal condition is that the number of samples is lower than n_{\min} . We remedy this problem with the help of a new parameter V_{\min} which specifies the minimal variance between prediction and measure necessary to allow splitting. A naive implementation of Extra-Trees leads to a second problem: it may generate nodes with very few samples, which paves the way to overfitting and is bad for linear interpolation. Therefore, we changed the choice of the split values. Instead of choosing it uniformly from the minimum to the maximum of the samples, our algorithm choose it uniformly between the n_{\min} -th smallest and highest values, which guarantees that each node of the split tree contains at least n_{\min} samples.

4.3 Projection of a regression tree

Let consider a tree $t : S \times A \mapsto \mathbb{R}$, we can define the projection of the tree t on the state s as another tree $\mathcal{P}(t, s) = t' : A \mapsto \mathbb{R}$. Since s is known, t' does not contain any split depending on s value and therefore contains only splits related to the action space. It is easy to create a hyperrectangle \mathcal{H} corresponding to state s .

$$\mathcal{H}(s) = \begin{pmatrix} s_1 & s_1 \\ \vdots & \vdots \\ s_{D_s} & s_{D_s} \\ \min(A_1) & \max(A_1) \\ \vdots & \vdots \\ \min(A_{D_A}) & \max(A_{D_A}) \end{pmatrix}$$

The pseudo-code for tree projection is shown in Algorithm 1.

Algorithm 1 The tree projection algorithm

```

1: function PROJECTTREE( $t, \mathcal{H}$ )
2:   return projectNode(root( $t$ ),  $\mathcal{H}$ )
3: end function
4: function PROJECTNODE( $node, \mathcal{H}$ )
5:   if isLeaf( $node$ ) then
6:     return  $node$ 
7:   end if
8:    $d \leftarrow$  splitDim( $node$ )
9:    $v \leftarrow$  splitVal( $node$ )
10:  if  $v > \mathcal{H}_{d,M}$  then
11:     $node \leftarrow$  projectTree(LC( $node$ ),  $\mathcal{H}$ )
12:  else if  $v \leq \mathcal{H}_{d,m}$  then
13:     $node \leftarrow$  projectTree(UC( $node$ ),  $\mathcal{H}$ )
14:  else
15:    LC( $node$ )  $\leftarrow$  projectTree(LC( $node$ ),  $\mathcal{H}$ )
16:    UC( $node$ )  $\leftarrow$  projectTree(UC( $node$ ),  $\mathcal{H}$ )
17:  end if
18:  return  $node$ 
19: end function

```

4.4 Weighted average of regression trees

Let t_1 and t_2 be two regressions trees mapping X to Y , weight respectively by w_1 and w_2 , we define the weighted average of the trees as a tree $t' = \mu(t_1, t_2, w_1, w_2)$ such as:

$$\forall x \in X, t'(x) = \frac{t_1(x)w_1 + t_2(x)w_2}{w_1 + w_2}$$

A simple scheme for computing t' would be to root a replicate of t_2 at each leaf of t_1 . However this would lead to an overgrown tree containing various 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 $x_1 \leq 2$.

Therefore, we designed an algorithm which merges the two trees by walking simultaneously both trees from the root to the leaves, and performing on-the-fly optimizations. The algorithm pseudo-code is shown in Algorithm 2. An example of input and output of the algorithm is shown in Figure 3. By this way, we also tend to keep an original aspect of the regression tree which is that the top-most nodes carry the most important splits (i.e. splits that strongly reduce the variance of their inner sets of samples).

4.5 Pruning trees

Although our merging procedure helps to reduce the size of the final trees, the combination of M trees might still lead to a tree of size $O(|t|^M)$. Therefore we developed a pruning algorithm which aims at removing the split nodes which bring the smallest change to the prediction function. The only nodes that the algorithms is allowed to remove are nodes that are parent of two leaves. We define the *loss* \mathcal{L} to the prediction function for a node n concerning a hyperrectangle \mathcal{H}_n as:

$$\mathcal{L} = \int_{x \in \mathcal{H}_l} (\phi'(x) - \phi_l) dx + \int_{x \in \mathcal{H}_u} (\phi'(x) - \phi_u) dx \quad (3)$$

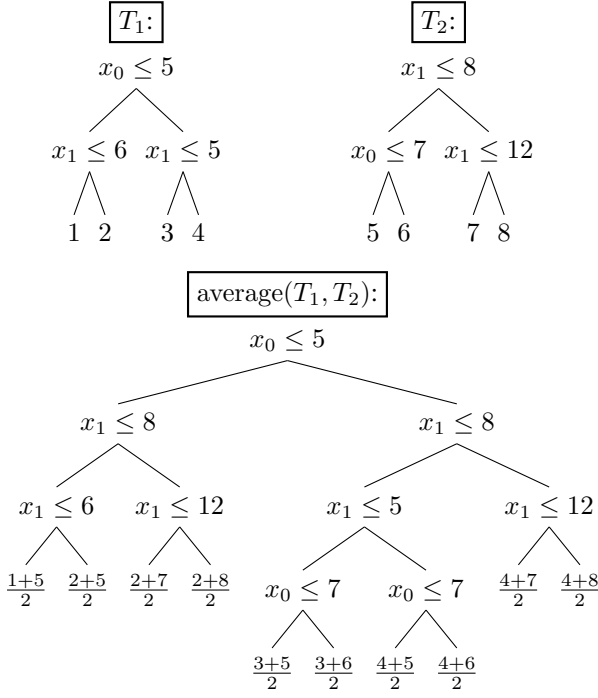


Figure 3: An example of tree merge

Algorithm 2 The averaging tree algorithm

```

1: function AVGTREES( $t', t_1, t_2, w_1, w_2, \mathcal{H}$ )
2:   avgNodes(root( $t'$ ), root( $t_1$ ), root( $t_2$ ),  $w_1, w_2, \mathcal{H}$ )
3: end function
4: function AVGNODES( $n', n_1, n_2, w_1, w_2, \mathcal{H}$ )
5:   if isLeaf( $n_1$ ) then
6:     if isLeaf( $n_2$ ) then
7:        $\phi_{n'} = \frac{w_1 \phi_{n_1} + w_2 \phi_{n_2}}{w_1 + w_2}$ 
8:     else
9:       avgNodes( $n', n_2, n_1, w_2, w_1, \mathcal{H}$ )
10:    end if
11:   else
12:      $d \leftarrow \text{splitDim}(n_1)$ 
13:      $v \leftarrow \text{splitVal}(n_1)$ 
14:      $v_m \leftarrow \mathcal{H}_{d,m}$ 
15:      $v_M \leftarrow \mathcal{H}_{d,M}$ 
16:     if  $v_M \leq v$  then
17:       avgNodes( $n', n_2, \text{LC}(n_1), w_2, w_1, \mathcal{H}$ )
18:     else if  $v_m < v$  then
19:       avgNodes( $n', n_2, \text{UC}(n_1), w_2, w_1, \mathcal{H}$ )
20:     else
21:       split( $n'$ )  $\leftarrow$  split( $n$ )
22:        $\mathcal{H}_{d,M} \leftarrow v$ 
23:       avgNodes( $\text{LC}(n'), n_2, \text{LC}(n_1), w_2, w_1, \mathcal{H}$ )
24:        $\mathcal{H}_{d,M} \leftarrow v_M$ 
25:        $\mathcal{H}_{d,m} \leftarrow v$ 
26:       avgNodes( $\text{UC}(n'), n_2, \text{UC}(n_1), w_2, w_1, \mathcal{H}$ )
27:        $\mathcal{H}_{d,m} \leftarrow v_m$ 
28:     end if
29:   end if
30: end function

```

Where l and u are the lowerchild and upperchild of n respectively, and:

$$\phi' = \frac{|\mathcal{H}_u| \phi_u + |\mathcal{H}_l| \phi_l}{|\mathcal{H}|} \quad (4)$$

The prediction function ϕ' given by equation 4 is a weighted average of the prediction functions of both children weighted by the size of the space concerned by each one. This choice reduces the impact of the prediction on a leaf when merged with a bigger leaf. Our definition of the loss \mathcal{L} in equation 3 also considers the size of the spaces since we compute the integral. The main interest of this method is to reduce the average error on the whole tree by weighting the cost of an error by the size of its space. While most pruning procedures in litterature are centered about reducing the risk of overfitting, our algorithm (Algorithm 3) cares only about reducing the size of the tree, ensuring that the complexity of the representation does not go above a given threshold. Since this procedure is not based on the training set used to grow the forest, it is not necessary to have an access to the training set in order to prune the tree. When merging the M trees of a forest, it is crucial to prune the tree resulting of two merge before applying another merge.

Algorithm 3 The tree pruning algorithm

```

1: splits = {} ▷ Map from (node,  $\mathcal{L}$ ) to  $\phi$ , ordered by  $\mathcal{L}$ 
2: for all  $n \in \text{preLeaves}(t)$  do
3:    $\mathcal{L} = \text{getLoss}(n)$  ▷ See Eq. 3
4:    $\phi = \text{getAverageFunction}(n)$  ▷ See Eq. 4
5:   add ( $(n, \mathcal{L}), \phi$ ) to splits
6: end for
7: nbLeaves  $\leftarrow$  countLeaves( $t$ )
8: while nbLeaves > maxLeaves do
9:   ( $(n, \mathcal{L}), \phi$ )  $\leftarrow$  popFirst(splits)
10:   $\phi_n \leftarrow \phi$ 
11:  removeChild( $n$ )
12:  if isLastSplit(father( $n$ )) then
13:     $n \leftarrow \text{father}(n)$ 
14:     $\mathcal{L} = \text{getLoss}(n)$  ▷ See Eq. 3
15:     $\phi = \text{getAverageFunction}(n)$  ▷ See Eq. 4
16:    add ( $(n, \mathcal{L}), \phi$ ) to splits
17:  end if
18:  nbLeaves  $\leftarrow$  nbLeaves - 1
19: end while

```

5 Approximation of the optimal policy

In this section, we propose three new methods used to choose optimal action for a given state based on an estimation of the Q -value by a regression forest. While learning of the policy can be computationally demanding since it is performed offline, it is crucial to obtain descriptions of the policies that allow very quick computation of the action, given the current state.

5.1 Learning the continuous policy

In order to compute the best policy given an approximation of the Q -value \hat{Q} by a regression forest F , we need to solve the following equation:

$$\hat{\pi}^*(s) = \operatorname{argmax}_{a \in A} F((s, a)) \quad (5)$$

Given s , the most straightforward way to compute $\hat{\pi}^*(s)$ consists in merging all the trees of F projected on s into a single tree t' . Since the size of t' can grow exponentially with the number of trees, we compute an approximation of t' , denoted \hat{t}' by imposing a limit on the number of leaves using Algorithm 3. Then it is possible to approximate the best actions by simply iterating on all the leaves of \hat{t}' and computing the maximum of the function ϕ of the leaf in its interval. While this solution does not provide the exact policy which would be induced by F , it provides a roughly good approximation. We refer to this method by *Fitted Q-Iteration*, *FQI* for short.

The FQI is computationally too expensive to be used in online situation: the computation of a single action requires exploring a potentially large number of leaves. Therefore, in order to provide a very quick access to the optimal action for a given state, we propose a new scheme. By decomposing the policy function $\pi : S \mapsto A$ into several functions $\pi_j : S \mapsto A_j$ where j is a dimension of the action space, we can easily generate samples and use them to train regression forests which provide estimates of the policy for each dimension. We named this process *Fitted Policy Forest* and abbreviate it by FPF. We use two variants, one using a piecewise constant model for the nodes, PWC for short, and another using piecewise linear model for the nodes, PWL for short. We refer to these two methods by *FPF:PWC* and *FPF:PWL* respectively. Policies resulting of the FPF algorithm provides a quick access. If such a policy is composed of M trees with a maximal number of nodes n , the complexity of getting the action is $O(M \log(n))$. Since the values used for M does not need to be high to provide a good approximation (Ernst, Geurts, and Wehenkel 2005), this complexity makes FPF perfectly suited for real-time applications where online computational resources are very limited, such as robotics.

6 Exploration

While MRE (Nouri and Littman 2009) provide a strong basis to build exploration algorithm, we found that its performance can be strongly improved by bringing three modifications. First we change the equation used to compute the knownness, second we use bagging technic to improve the estimation of the knownness, and third we modify the rule used for Q -value update.

6.1 Original definition

Multi Resolution Exploration (Nouri and Littman 2009) propose 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 estimate the degree of knowledge of a couple $(s, a) \in S \times A$. During the execution

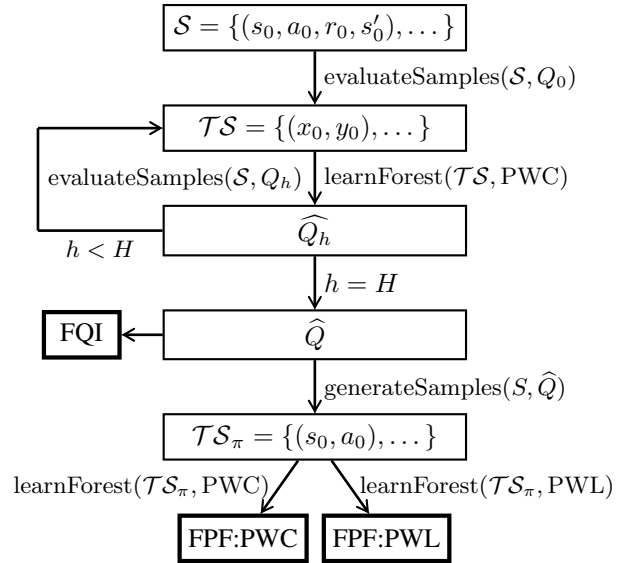


Figure 4: A flowchart of the different methods

of the algorithm, when action a is taken in state s , a point $p = (s_1, \dots, s_{\dim S}, a_1, \dots, a_{\dim A})$ 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 the following equation:

$$\kappa(p) = \min \left(1, \frac{|P|}{\nu} \frac{\frac{1}{\sqrt{\frac{k}{nk/\nu}}}}{\|\mathcal{H}\|_\infty} \right) \quad (6)$$

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 $\mathcal{H} = \text{space}(\text{leaf}(\tau, p))$. A crucial point of this equation is the fact that the knownness value depends on three main aspects: 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 splitted into two leafs l_1 and l_2 of the same size, and the dimension is chosen using a round-robin. Then the points stored in l_0 are attributed to l_1 and l_2 depending on their value.

MRE also changes the update rule by using an optimistic rule which replace equation (2) by equation (7):

$$y' = \kappa(s, a)y + (1 - \kappa(s, a)) \frac{R_{\max}}{1 - \gamma} \quad (7)$$

where R_{\max} is the maximal reward which can be awarded in a single step and y is the result obtained by equation (2). This update 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 $1 - \kappa(s, a)$.

6.2 Computation of the knownness value

Initial definition of the knownness is given at Equation (6). Since this definition does only depend on the biggest dimension, we have the following. Consider a leaf l_0 with a knownness τ_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 $k_0 > k_2$. This leads to the unnatural fact that adding a point in the middle of other points can decrease the knowledge of all these points.

We decide to base our knowledge on the ratio between the density of points inside the leaf and the density of points. Thus replacing Equation (6) by Equation (8):

$$\kappa(p) = \min \left(1, \frac{\frac{|\text{points}(\text{leaf}(\tau, p))|}{|\text{leaf}(\tau, p)|}}{\frac{n}{|S \times A|}} \right) \quad (8)$$

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.

6.3 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, Ernst, and Wehenkel 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 extra-trees.

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.

6.4 Modification of the Q -value update

The Q -value update rule proposed by MRE improve 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. Therefore, even if for a state s and an action a , $\kappa(s, a) \approx 0$, it might have no influence at all.

In order to solve this issue, we decided to avoid the modification of the training set creation, thus using Equation (2). In place of modifying those samples, we simply update the regression forest by applying the following modifcator on every leaf of every tree:

$$v' = v\kappa(c) + R_{\max}(1 - \kappa(c)) \quad (9)$$

with c the center of the leaf, v the original value and v' the new value.

7 Experimental results

We present experimental results under two different learning setup. First, the results obtained by FPF in a batch reinforcement learning, second, the performances obtained by combining MRE and FPF for online learning.

7.1 Batch reinforcement learning

We used three benchmark problems classical in RL 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 datas. Thus, the graphical lines shown for BAS are approximative 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 perfectly suited for high-frequency control in embedded systems.

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. 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:

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

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, Tanaka, and Griffin 1996):

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

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

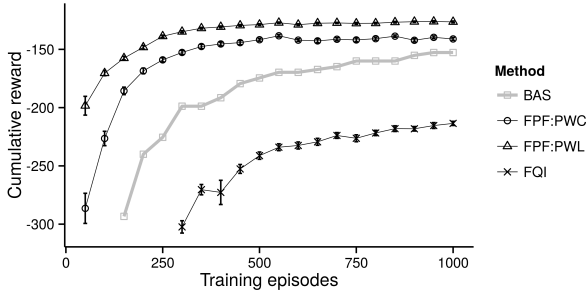


Figure 5: Performance on the Inverted Pendulum Stabilization problem

and using low forces to balance the inverted pendulum are rated higher than others.

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 5 and average their cumulative reward (vertical bars denote 95% confidence interval). The parameters used to produce the policies are shown in Table 1.

Learning a policy from the Q -value tree clearly outperform a direct use 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 forest to learn the policy from the Q -value can lead to drastical 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 provide 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 tend to slightly overfit the data, additionally, it uses PWC approximation. Therefore, using it directly lead to an important quantization noise. Using a large value for n_{\min} might be seen as applying a smoothing, which is considered as necessary for regression trees sampling a stochastic function according to (Ernst, Geurts, and Wehenkel 2005). The need for a large number of samples is increased for FPF:PWL, because providing an accurate linear interpolation of a noisy application requires a lot of samples.

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, Sutton, and Ram 1997). The double integrator is a linear dynamics system where the aim of the controller

Table 1: Parameters used for Inverted Pendulum Stabilization

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

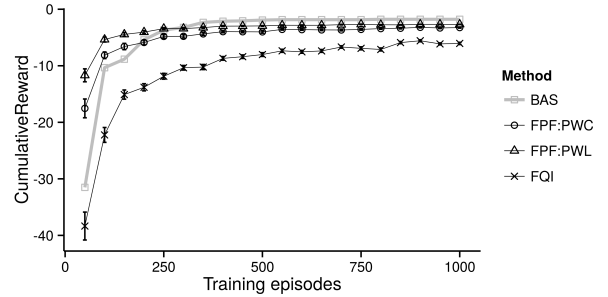


Figure 6: Performance on the Double Integrator problem

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 $a \in [-1, 1]$ 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 $\gamma = 0.98$.

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 6 and average their results. The parameters used for learning the policy are shown in Table 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, a polynome is used for Least-Square Policy Iteration in (Pazis and Lagoudakis 2009), fitting the fact that the optimal policy is known to be a linear-quadratic regulator (Santamaria, Sutton, and Ram 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, Geurts, and Wehenkel 2005) which was also used as a benchmark in (Pazis and Lagoudakis 2009).

Table 2: Parameters used for Double Integrator

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

In this problem an underactuated car must reach the top of a hill. The state 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 $u \in [-4, 4]$. 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.

It is well known that the solution to this problem is a bang-bang strategy, i.e. a nearly optimal strategy exists which uses only the set of actions $\{-4, 4\}$. As stated in (Pazis and Lagoudakis 2009), this problem is one of the worst case for reinforcement learning with continuous action space, since it requires to learn a binary strategy composed of actions which have not been sampled frequently. It has been shown in (Ernst, Geurts, and Wehenkel 2005) that introducing more actions usually reduce the performance of the controller. Therefore, we do not hope to reach a performance comparable to those achieved with a binary choice. This benchmark is more aimed to assess the performance of our algorithms, in one of the worst case.

While the sample of the two previous algorithms are based on episodes generated at a starting point, the samples used for the *Car on the hill* problem are generate by sampling uniformly the state and action spaces. This procedure is the same which has been used in (Ernst, Geurts, and Wehenkel 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 $(-0.5, 0)$.

We show the histogram of the number of steps required for each method at Figure 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 take 20 to 25 steps on average while it is mentioned in (Pazis and Lagoudakis 2009) that BAS controller requires 20 to 45 steps on average. Over the six hundred of experiments gathered across three different methods, the maximal number of steps measured was 33. Therefore, we can consider that our results strongly outperforms BAS results.

Car on the Hill is the only problem on which we have

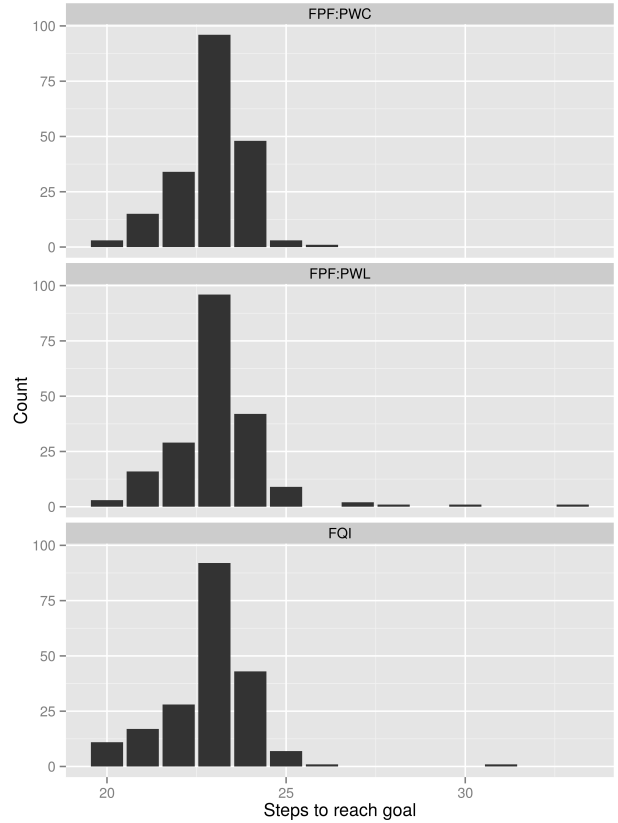


Figure 7: Performance on the Car on the Hill problem

not experienced significant difference between FPF and FQI. Since one of the main advantage of FPF approach is to reduce the quantization noise of the FQI method, this result is logical. 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. Therefore, we can affirm that FPF is highly preferable to FQI on this problem.

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 8 and the average time spent for learning the policies in 9. Experiments were runned 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” problem due to the lack of space, similar results were observed for the two other problems.

It is important to note that the cost displayed in Figure 8

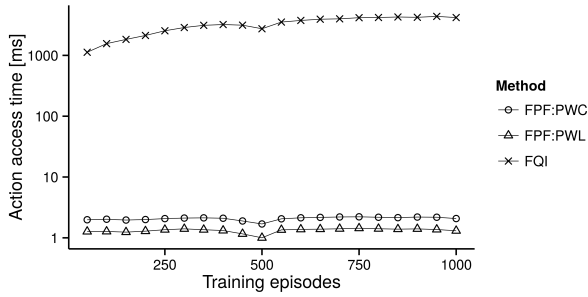


Figure 8: Evaluation time by episode for the Double Integrator

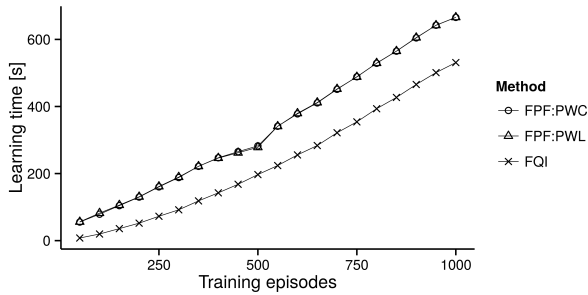


Figure 9: Learning time by episode for the Double Integrator

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 policies 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 9. Therefore, when it is possible to use FQI, it should also be possible to use FPF without increasing too much the offline cost.

7.2 Online reinforcement learning

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 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 *trial* a whole execution of the MRE algorithm on the problem.

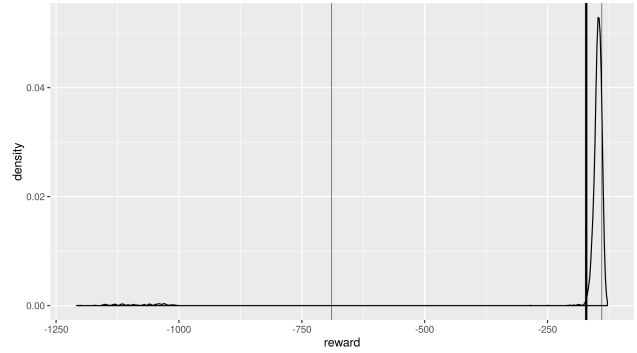


Figure 10: Reward repartition for online learning on Inverted Pendulum Stabilization

Inverted pendulum stabilization This problem is exactly the same as defined in Section 7.1, but it is used in a context of online reinforcement learning. The result presented in this section represent 10 trials of 100 episodes. Each trial 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 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 maximal reward 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 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.

Inverted pendulum swing-up For this problem, instead of using a mathematical model, we decided to use the simulator Gazebo² and to control it using ROS³. 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 RL methods in robotics. We developed a simple model composed of a support and a pendulum which are bounded by

²<http://gazebosim.org>

³<http://www.ros.org>

an angular joint. The angular joint is controlled in torque and is underactuated, i.e. the available torque is not sufficient to maintain the pendulum in an horizontal state. 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 $0.1[Nm.s/rad]$, the friction coefficient is $0.1[Nm]$, the maximal torque is $\tau_{\max} = 15[Nm]$, the maximal angular speed is $\dot{\theta}_{\max} = 10[rad/s]$ and the control frequency is $10[Hz]$. The reward function used is the following

$$r = - \left(\left\| \frac{\theta}{\pi} \right\| + \left(\frac{\tau}{\tau_{\max}} \right)^2 \right) \quad (10)$$

Where θ is the angular position of the pendulum (0 denote an upward position), and τ represent the torque applied on the axis. If $\|\dot{\theta}\| > \dot{\theta}_{\max}$, 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 scale of action, large actions in order to inject energy in the system and fine action in order to stabilize the system.

The result presented in this section represent 5 trials of 100 episodes. Each trial 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 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 the best and the worst policy. We exhibit the trajectory of the best and worst evaluation episode in Figure 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.

8 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 piecewise linear function to represent the Q -value often leads to divergence as mentioned in (Ernst, Geurts, and Wehenkel 2005), the same problem did not appear on 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 method. 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

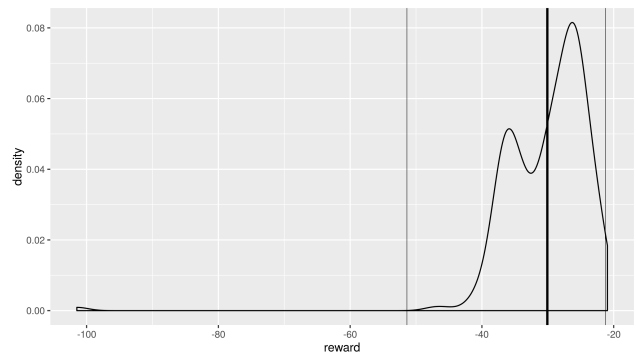


Figure 11: Reward repartition for online learning on Inverted Pendulum Swing-Up

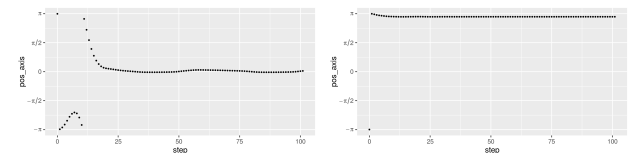


Figure 12: Best and worst episode for Inverted Pendulum Swing-Up

of the problem, 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.

9 Conclusion

This article introduces Fitted Policy Forest, an algorithm extracting a policy from a regression forest representing the Q -value. FPF presents several advantages: it has an extremely low computational cost to access the optimal action, it does not require expert knowledge about the problem, it is particularly successful at solving problems requiring fine actions in stochastic problems and it can be used with any algorithm producing regression forests. The effectiveness of our algorithm in a batch setup is demonstrated in three different benchmarks. The use of FPF in online reinforcement learning is also discussed and assessed by using MRE as an exploration strategy. Experimental results suggest that exploration can lead to satisfying results without requiring any tuning on the parameters. In the future, we also would like to apply this approach to closed-loop control of Robocup humanoid robots.

References

- Behnke, S. 2006. Online trajectory generation for omnidirectional biped walking. *Proceedings - IEEE International Conference on Robotics and Automation 2006*(May):1597–1603.
- Breiman, L. 1996. Bagging predictors. *Machine Learning* 24(2):123–140.
- Busoniu, L.; Babuska, R.; Schutter, B. D.; Ernst, D.; Busoniu, L.; Babuska, R.; Schutter, B. D.; and Ernst, D. 2010. Reinforcement learning and dynamic programming using function approximators. 260.
- Ernst, D.; Geurts, P.; and Wehenkel, L. 2005. Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research* 6(1):503–556.
- Geurts, P.; Ernst, D.; and Wehenkel, L. 2006. Extremely randomized trees. *Machine Learning* 63(1):3–42.
- Koenig, S., and Simmons, R. G. 1996. The effect of representation and knowledge on goal-directed exploration with reinforcement-learning algorithms. *Machine Learning* 22(1-3):227–250.
- Li, L.; Littman, M. L.; and Mansley, C. R. 2009. Online exploration in least-squares policy iteration. *The 8th International Conference on Autonomous Agents and Multiagent Systems* 733–739.
- Li, K.-C.; Lue, H.-H.; and Chen, C.-H. 2000. Interactive Tree-Structured Regression via Principal Hessian Directions. *Journal of the American Statistical Association* 95(450):547–560.
- Loh, W.-Y. 2011. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 1(1):14–23.
- Nouri, A., and Littman, M. L. 2009. Multi-resolution Exploration in Continuous Spaces. *Advances in Neural Information Processing Systems* 1209–1216.
- Pazis, J., and Lagoudakis, M. G. 2009. Binary action search for learning continuous-action control policies. *Proceedings of the 26th International Conference on Machine Learning (ICML)* 793–800.
- Peters, J., and Schaal, S. 2008. Reinforcement learning of motor skills with policy gradients. *Neural Networks* 21(4):682–697.
- Preparata, F. P., and Shamos, M. I. 1985. *Computational geometry: an introduction*, volume 47.
- Puterman, M. L. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*.
- Sanner, S.; Delgado, K. V.; and de Barros, L. N. 2012. Symbolic Dynamic Programming for Discrete and Continuous State MDPs. In *Proceedings of the 26th Conference on Artificial Intelligence*, volume 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.
- 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.
- Weinstein, A., and Littman, M. 2013. Open-Loop Planning in Large-Scale Stochastic Domains. In *27th AAAI Conference on Artificial Intelligence*, volume 1, 1436–1442.
- Weinstein, A. 2014. *Local Planning for Continuous Markov Decision Processes*. Ph.D. Dissertation, The State University of New Jersey.
- Zamani, Z.; Sanner, S.; and Fang, C. 2012. Symbolic Dynamic Programming for Continuous State and Action MDPs.