Fast Reinforcement Learning with Large Action Sets using Error-Correcting Output Codes for MDP Factorization

Gabriel Dulac-Arnold[1], Ludovic Denoyer[1], Philippe Preux[2], and Patrick Gallinari[1]

[1] LIP6–UPMC, Case 169 4 Place Jussieu,
Paris 75005, France
`firstname.lastname@lip6.fr`

[2] LIFL (UMR CNRS) & INRIA Lille Nord-Europe – Université de Lille
Villeneuve d'Ascq, France
`firstname.lastname@inria.fr`

Abstract. The use of Reinforcement Learning in real-world scenarios is strongly limited by issues of scale. Most RL learning algorithms are unable to deal with problems composed of hundreds or sometimes even dozens of possible actions, and therefore cannot be applied to many real-world problems. We consider the RL problem in the supervised classification framework where the optimal policy is obtained through a multiclass classifier, the set of classes being the set of actions of the problem. We introduce error-correcting output codes (ECOCs) in this setting and propose two new methods for reducing complexity when using rollouts-based approaches. The first method consists in using an ECOC-based classifier as the multiclass classifier, reducing the learning complexity from $\mathcal{O}(A^2)$ to $\mathcal{O}(A\log(A))$. We then propose a novel method that profits from the ECOC's coding dictionary to split the initial MDP into $\mathcal{O}(\log(A))$ separate two-action MDPs. This second method reduces learning complexity even further, from $\mathcal{O}(A^2)$ to $\mathcal{O}(\log(A))$, thus rendering problems with large action sets tractable. We finish by experimentally demonstrating the advantages of our approach on a set of benchmark problems, both in speed and performance.

1 Introduction

The goal of Reinforcement Learning (RL) and more generally sequential decision making is to learn an optimal policy for performing a certain task within an environment, modeled by a Markov Decision Process (MDP). In RL, the dynamics of the environment are considered as unknown. This means that to obtain an optimal policy, the learner interacts with its environment, observing the outcomes of the actions it performs. Though well understood from a theoretical point of view, RL still faces many practical issues related to the complexity of the environment, in particular when dealing with large state or action sets. Currently, using function approximation to better represent and generalize over the environment is a common approach for dealing with large *state* sets. However, learning with large *action* sets has been less explored and remains a key challenge.

When the number of possible actions A is neither on the scale of 'a few' nor outright continuous, the situation becomes difficult. In particular cases where the action space is continuous (or nearly so), a regularity assumption can be made on the consequences of the actions concerning either a certain smoothness or Lipschitz property over the action space [1, 2, 3]. However, situations abound in which the set of actions is discrete, but the number of actions lies somewhere between 10 and 10^4 (or greater) — Go, Chess, and planning problems are among these. In the common case where the action space shows no regularity *a priori*, it is not possible to make any assumptions regarding the consequence of an action that has never been applied.

In this article, we present an algorithm which can intelligently sub-sample even completely irregular action spaces. Drawing from ideas used in multiclass supervised learning, **we introduce a novel way to significantly reduce the complexity of learning (and acting) with large action sets**. By assigning a multi-bit code to each action, we create binary clusters of actions through the use of *Error Correcting Output Codes* (ECOCs) [4]. Our approach is anchored in Rollout Classification Policy Iteration (RCPI) [5], an algorithm well know for its efficiency on real-world problems. We begin by proposing a simple way to reduce the computational cost of any policy by leveraging the clusters of actions defined by the ECOCs. We then extend this idea to the problem of learning, and propose a new RL method that allows one to find an approximated optimal policy by solving a set of 2-action MDPs. While our first model — ECOC-extended RCPI (ERCPI) — reduces the overall learning complexity from $\mathcal{O}(A^2)$ to $\mathcal{O}(A\log(A))$, our second method — Binary-RCPI (BRCPI) — reduces this complexity even further, to $\mathcal{O}(\log(A))$.

The paper is organized as follow: We give a brief overview of notation and RL in Section 2.1, then introduce RCPI and ECOCs in Sections 2.2 and 2.3 respectively. We present the general idea of our work in Section 3. We show how RCPI can be extended using ECOCs in 3.2, and then explain in detail how an MDP can be factorized to accelerate RCPI during the learning phase in 3.3. An in-depth complexity analysis of the different algorithms is given in Section 3.4. Experimental results are provided on two problems in Section 4. Related work is presented in Section 5.

2 Background

In this section, we cover the three key elements to understanding our work: Markov Decision Problems, Rollout Classification Policy Iteration, and Error-Correcting Output Codes.

2.1 Markov Decision Process

Let a Markov Decision Process $\mathcal{M}$ be defined by a 4-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R)$.

- $\mathcal{S}$ is the set of possible states of the MDP, where $s \in \mathcal{S}$ denotes one state of the MDP.
- $\mathcal{A}$ is the set of possible actions, where $a \in \mathcal{A}$ denotes one action of the MDP.
- $T : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the MDP's transition function, and defines the probability of going from state s to state s' having chosen action a: $T(s', s, a) = P(s'|s, a)$.
- $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is a reward function defining the expected immediate reward of taking action a in state s. The actual immediate reward for a particular transition is denoted by r.

In this article, we assume that the set of possible actions is the same for all states, but our work is not restricted to this situation; the set of actions can vary with the state without any drawbacks.

Let us define a policy, $\pi : \mathcal{S} \rightarrow \mathcal{A}$, providing a mapping from states to actions in the MDP. In this paper, without loss of generality, we consider that the objective to fulfill is the optimization of the expected sum of γ-discounted rewards from a given set of states D: $J_\pi(s) = \mathbb{E}[\sum_{k \geq 0} \gamma^k r_{t+k} | s_t = s \in D, \pi]$.

A policy's performance is measured w.r.t. the objective function J_π. The goal of RL is to find an optimal policy π^* that maximizes the objective function: $\pi^* = argmax_\pi J_\pi$.

In an RL problem, the agent knows both $\mathcal{S}$ and $\mathcal{A}$, but is not given the environment's dynamics defined by T and R. In the case of our problems, we assume that the agent may start from any state in the MDP, and can run as many simulations as necessary until it has learned a good policy.

2.2 Rollout Classification Policy Iteration

We anchor our contribution in the framework provided by RCPI [5]. RCPI belongs to the family of Approximate Policy Iteration (API) algorithms, iteratively improving estimates of the Q-function — $Q_\pi(s, a) = \mathbb{E}[J_\pi(s)|\pi]$. In general, API uses a policy π to estimate Q through simulation, and then approximates it by some form of regression on the estimated values, providing $\tilde{Q}_\pi$. This is done first with an initial (and often random) policy π_0, and is iteratively repeated until $\tilde{Q}_\pi$ is properly estimated. $\tilde{Q}_\pi(s, a)$ is estimated by running K *rollouts* i.e. Monte-Carlo simulations using π to estimate the expected reward. The new policy π' is thus the policy that chooses the action with the highest $\tilde{Q}_\pi$-value for each state.

In the case of RCPI, instead of using a function approximator to estimate $\tilde{Q}_\pi$, the best action for a given s is selected using a classifier, without explicitly approximating the Q-value. This estimation is usually done using a binary classifier f_θ over the state-action space such that the new policy can be written as:

$$\pi'(s) = \operatorname*{argmax}_{a \in \mathcal{A}_s} f_\theta(s, a). \tag{1}$$

The classifier's training set $\mathcal{S}_{\mathrm{T}}$ is generated through Monte-Carlo sampling of the MDP, estimating the optimal action for each state sampled. Once generated, these optimal state-action pairs (s, a) are used to train a supervised classifier;

the state is interpreted as the feature vector, and the action a as the state's label. In other words, RCPI is an API algorithm that uses Monte Carlo simulations to transform the RL problem into a multiclass classification problem.

2.3 Error-Correcting Output Codes

In the domain of multiclass supervised classification in large label spaces, ECOCs have been in use for a while [4]. We will cover ECOCs very briefly here, as their adaptation to an MDP formalism is well detailed in Section 3.2.

Given a multiclass classification task with a label set $\mathcal{Y}$, the $|\mathcal{Y}|$ class labels can be encoded as binary integers using as few as $C = \log_2(|\mathcal{Y}|)$ bits. ECOCs for classification assume that each label is associated to a binary code of length[3] $C = \gamma \log(|\mathcal{Y}|)$ with $\gamma \geq 1$.

The main principle of multiclass classifiers with ECOCs is to learn to predict the output code instead of directly predicting the label, transforming a supervised learning problem with $|\mathcal{Y}|$ classes into a set of $C = \gamma \log(|\mathcal{Y}|)$ binary supervised learning problems. Once trained, the class of a datum x can be inferred by passing the datum to all the classifiers and concatenating their output into a predicted label code: $code(x) = (f_{\theta_0}(x), \cdots, f_{\theta_C}(x))$. The predicted label is thus the label with the closest code in terms of Hamming distance. As a side note, Hamming distance look-ups can be done in logarithmic time by using tree-based approaches such as k-d trees [6]. ECOCs for classification can thus infer with a complexity of $\mathcal{O}(log(|\mathcal{Y}|)$.

3 Extended & Binary RCPI

We separate this paper's contributions into two parts, the second part building on the first one. We begin by showing how ECOCs can be easily integrated into a classifier-based policy, and proceed to show how the ECOC's coding matrix can be used to factorize RCPI into a much less complex learning algorithm.

3.1 General Idea

The general idea of our two algorithms revolves around the use of ECOCs for representing the set of possible actions, $\mathcal{A}$. This approach assigns a multi-bit code of length $C = \gamma \log(A)$ to each of the A actions. The codes are organized in a *coding matrix*, illustrated in Figure 1 and denoted $\mathbf{M}^c$. Each row corresponds to one action's binary code, while each column is a particular dichotomy of the action space corresponding to that column's associated bit b_i. In effect, each column is a projection of the A-dimensional action space into a 2-dimensional binary space. We denote $\mathbf{M}^c_{[a,*]}$ as the a^{th} row of $\mathbf{M}^c$, which corresponds to a's binary code. $\mathbf{M}^c_{[a,i]}$ corresponds to bit b_i of action a's binary code.

[3] Different methods exist for generating such codes. In practice, it is customary to use redundant codes where $\gamma \approx 10$.

	b_1	b_2	b_3
a_1	$+$	$+$	$-$
a_2	$-$	$+$	$-$
a_3	$+$	$-$	$+$
a_4	$-$	$+$	$+$
a_5	$+$	$-$	$-$

Fig. 1: An example of a 5-actions, $C = 3$-bits coding matrix. The code of action 1 is $(+, +, -)$.

Our main idea is to consider that each bit corresponds to a binary sub-policy denoted π_i. By combining these sub-policies, we can derive the original policy π one wants to learn as such:

$$\pi(s) = \operatorname*{argmin}_{a \in \mathcal{A}} d_{\mathrm{H}}(\mathbf{M}^c_{[a,*]}, (\pi_1(s), \cdots, \pi_C(s)), \tag{2}$$

where $\mathbf{M}^c_{[a,*]}$ is the binary code for action a, and d_{H} is the Hamming distance. For a given a state s, each sub-policy provides a binary action $\pi_i(s) \in \{-, +\}$, thus producing a binary vector of length C. $\pi(s)$ chooses the action a with the binary code that has the smallest Hamming distance to the concatenated output of the C binary policies.

We propose two variants of RCPI that differ by the way they learn these sub-policies. ECOC-extended RCPI (ERCPI) replaces the standard definition of π by the definition in Eq. (2), both for learning and action selection. The Binary-RCPI method (BRCPI) relaxes the learning problem and considers that all the sub-policies can be learned independently on separate binary-actioned MDPs, resulting in a very rapid learning algorithm.

3.2 ECOC-Extended RCPI

ERCPI takes advantage of the policy definition in Equation (2) to decrease RCPI's complexity. The C sub-policies — $\pi_{i \in [1,C]}$ — are learned simultaneously on the original MDP, by extending the RCPI algorithm with an ECOC-encoding step, as described in Algorithm 1. As any policy improvement algorithm, ERCPI iteratively performs the following two steps:

Simulation Step: This consists in performing Monte Carlo simulations to estimate the quality of a set of state-action pairs. From these simulations, a set of training examples $\mathcal{S}_{\mathrm{T}}$ is generated, in which data are states, and labels are the estimated best action for each state.

Learning Step: For each bit b_i, $\mathcal{S}_{\mathrm{T}}$ is used to create a binary label training set $\mathcal{S}^i_{\mathrm{T}}$. Each $\mathcal{S}^i_{\mathrm{T}}$ is then used to train a classifier f_{θ_i}, providing sub-policy π'_i as in Eq. (1). Finally, the set of C sub-policies are combined to provide the final improved policy as in Eq. (2).

ERCPI's training algorithm is presented in Alg. 1.

The **Rollout** function used by ERCPI is identical to the one used by RCPI — π is used to estimate a certain state-action tuple's expected reward, $\tilde{Q}_\pi(s, a)$.

Algorithm 1: ERCPI

```
   Data:
   $\mathcal{S}_R$: uniformly sampled state set; $\mathcal{M}$: MDP; $\pi_0$: initial policy; $K$: number of
   trajectories; $T$: maximum trajectory length
1  $\pi = \pi_0$
2  repeat
3      $\mathcal{S}_\mathrm{T} = \emptyset$
4      foreach $s \in \mathcal{S}_R$ do
5          foreach $a \in A$ do
6              $\tilde{Q}_\pi(s,a) \leftarrow$ Rollout$(\mathcal{M}, s, a, K, \pi)$
7          end
8          $\mathcal{A}^* = \mathrm{argmax}_{a \in \mathcal{A}} \tilde{Q}_\pi(s,a)$
9          foreach $a^* \in \mathcal{A}^*$ do
10             $\mathcal{S}_\mathrm{T} = \mathcal{S}_\mathrm{T} \cup \{(s,a^*)\}$
11         end
12     end
13     foreach $i \in [1, C]$ do
14         $\mathcal{S}_\mathrm{T}^i = \emptyset$
15         foreach $(s,a) \in \mathcal{S}_T$ do
16             $a_i = \mathbf{M}^c_{[a,i]}$
17             $\mathcal{S}_\mathrm{T}^i = \mathcal{S}_\mathrm{T}^i \cup (s, a_i)$
18         end
19         $f_{\theta_i} =$ Train$(\mathcal{S}_\mathrm{T}^i)$
20         $\pi'_i$ from $f_{\theta_i}$ as defined in Eq. (1)
21     end
22     $\pi'$ as defined in Eq. (2)
23     $\pi = \alpha(\pi, \pi')$
24 until $\pi \sim \pi'$;
25 return $\pi$
```

Up to line 12 of Algorithm 1, ERCPI is in fact algorithmically identical to RCPI, with the slight distinction that only the best (s, a^*) tuples are kept, as is usual when using RCPI with a multiclass classifier.

ERCPI's main difference appears starting line 13; it is here that the original training set $\mathcal{S}_\mathrm{T}$ is mapped onto the C binary action spaces, and that each individual sub-policy π_i is learned. Line 16 replaces the original label of state s by its binary label in π_i's action space — this corresponds to bit b_i of action a's code.

The **Train** function on line 19 corresponds to the training of π_i's corresponding binary classifier on $\mathcal{S}_\mathrm{T}^i$. After this step, the global policy π' is defined according to Eq.(2). Note that, to ensure the stability of the algorithm, the new policy π obtained after one iteration of the algorithm is an alpha-mixture policy between the old π and the new π' obtained by the classifier (cf. line 23).

3.3 Binarized RCPI

ERCPI splits the policy improvement problem into C individual problems, but training still needs π, thus requiring the full set of binary policies. Additonnally, for each state, all A actions have to be evaluated by Monte Carlo simulation (Alg. 1, line 5). To reduce the complexity of this algorithm, we propose learning the C binary sub-policies — $\pi_{i\in[1,C]}$ — **independently**, transforming our initial MDP into C sub-MDPs, each one corresponding to the environment in which a particular π_i is acting.

Each of the π_i binary policies is dealing with its own particular representation of the action space, defined by its corresponding column in $\mathbf{M}^c$. For training, best-action selections must be mapped into this binary space, and each of the π_i's choices must be combined to be applied back in the original state space.

Let $\mathcal{A}_i^+, \mathcal{A}_i^- \subset \mathcal{A}$ be the action sets associated to π_i such that:

$$\begin{aligned}\mathcal{A}_i^+ &= \{a \in \mathcal{A} \mid \mathbf{M}^c_{[a,i]} = \text{`}+\text{'}\}\\ \mathcal{A}_i^- &= \mathcal{A} \setminus \mathcal{A}_i^+ = \{a \in \mathcal{A} \mid \mathbf{M}^c_{[a,i]} = \text{`}-\text{'}\}.\end{aligned} \tag{3}$$

For a particular i, $\mathcal{A}_i^+$ is the set of original actions corresponding to sub-action $+$, and $\mathcal{A}_i^-$ is the set of original actions corresponding to sub-action $-$.

We can now define C new binary MDPs that we name sub-MDPs, and denote $\mathcal{M}_{i\in[1,C]}$. They are defined from the original MDP as follows:

- $\mathcal{S}_i = \mathcal{S}$, the same state-set as the original MDP.
- $\mathcal{A}_i = \{+,-\}$.
- $T_i = T(s', s, a)P(a|a_i) = P(s'|s, a)P(a|a_i)$, where $P(a|a_i)$ is the probability of choosing action $a \in \mathcal{A}^{a_i}$, knowing that the sub-action applied on the sub-MDP $\mathcal{M}_i$ is $a_i \in \{+,-\}$. We consider $P(a|+)$ to be uniform for $a \in \mathcal{A}^+$ and null for $a \in \mathcal{A}^-$, and vice versa. $P(s'|s, a)$ is the original MDP's transition probability.
- $R_i(s, a_i) = \sum\limits_{a\in\mathcal{A}_i^{a_i}} P(a|a_i)R(s, a)$.

Each of these new MDPs represents the environment in which a particular binary policy π_i operates. We can consider each of these MDPs to be a separate RL problem for its corresponding binary policy.

In light of this, we propose to transform RCPI's training process for the base MDP into C new training processes, each one trying to find an optimal π_i for its corresponding $\mathcal{M}_i$. Once all of these binary policies have been trained, they can be used during inference in the manner described in Section 3.2.

The main advantage of this approach is that, since each of the $\gamma \log(A)$ sub-problems in Algorithm 2 is modeled as a binary-actioned MDP, increasing the number of actions in the original problem simply increases the number of sub-problems logarithmically, without increasing the complexity of these sub-problems – see Section 3.4.

Let us now discuss some details of BRCPI, as described in Algorithm 2. BRCPI resembles RCPI very strongly, except that instead of looping over the A

Algorithm 2: BRCPI

```
    Data:
    $\mathcal{S}_R$: uniformly sampled state set; $\mathcal{M}$: MDP; $\pi_0$: random policy; $K$: number of
    trajectories; $T$: maximum trajectory length; $C$: number of binary MDPs
 1  foreach $i \in C$ do
 2      $\pi_i = \pi_0$
 3      repeat
 4          $\mathcal{S}_{\mathrm{T}} = \emptyset$
 5          foreach $s \in \mathcal{S}_R$ do
 6              foreach $a \in \{+, -\}$ do
 7                  $\tilde{Q}_\pi(s, a) \leftarrow$ Rollout$(\mathcal{M}_i, s, a, K, \pi_i)$
 8              end
 9              $\mathcal{A}^* = \mathrm{argmax}_{a \in \mathcal{A}}\, \tilde{Q}_\pi(s, a)$
10              foreach $a^* \in \mathcal{A}^*$ do
11                  $\mathcal{S}_{\mathrm{T}} = \mathcal{S}_{\mathrm{T}} \cup \{(s, a^*)\}$
12              end
13          end
14          $f_{\theta_i} =$ Train$(\mathcal{S}_{\mathrm{T}})$
15          $\pi'_i$ from $f_{\theta_i}$ as defined in Eq. (1)
16          $\pi_i = \alpha(\pi_i, \pi'_i)$
17      until $\pi_i \sim \pi'_i$;
18      return $\pi$ as defined in Eq. (2)
19  end
```

actions on line 6, BRCPI is only sampling $\tilde{Q}$ for $+$ or $-$ actions. However, the inner loop is run $C = \gamma \log(A)$ times, as can be seen on line 1 of Algorithm 2.

Within the **Rollout** function (line 7), if π_i chooses sub-action '+', an action a_i from the original MDP is sampled from $\mathcal{A}_i^+$ following $P(a|a_i)$, and the MDP's transition function is called using this action. This effectively estimates the expected reward of choosing action $+$ in state s.

As we saw in Section 3.2, each $\mathcal{A}_i$ is a different binary projection of the original action set. Each of the π_i classifiers is thus making a decision considering a different split of the action space. Some splits may make no particular sense w.r.t. to the MDP at hand, and therefore the expected return of that particular π_i's $\mathcal{A}_i^+$ and $\mathcal{A}_i^-$ may be equal. This does not pose a problem, as that particular sub-policy will simply output noise, which will be corrected for by more pertinent splits given to the other sub-policies.

3.4 Computational Cost and Complexity

We study the computational cost of the proposed algorithms in comparison with the RCPI approach and present their respective complexities.

In order to define this cost, let us consider that $\mathcal{C}(S, A)$ is the time spent learning a multiclass classifier on S examples with A possible outputs, and $\mathcal{I}(A)$ is the cost of classifying one input.

The computational cost of one iteration of RCPI or ERCPI is composed of both a **simulation cost** — which corresponds to the time spent making Monte Carlo Simulation using the current policy — and a **learning cost** which corresponds to the time spent learning the classifier that will define the next policy[4]. This cost takes the following general form:

$$Cost = SAK \times T\mathcal{I}(A) + \mathcal{C}(S, A), \qquad (4)$$

where $T\mathcal{I}(A)$ is the cost of sampling one trajectory of size T, $SAK \times T\mathcal{I}(A)$ is the cost of executing the K Monte Carlo Simulations over S states testing A possible actions, and $\mathcal{C}(S, A)$ is the cost of learning the corresponding classifier[5].

The main difference between RCPI and ERCPI comes from the values of $\mathcal{I}(A)$ and $\mathcal{C}(S, A)$. When comparing ERCPI with a RCPI algorithm using a *one-vs-all* (RCPI-OVA) multiclass classifier — one binary classifier learned for each possible action — it is easy to see that our method reduces both $\mathcal{I}(A)$ and $\mathcal{C}(S, A)$ by a factor of $\frac{A}{\log A}$ — cf. Table 1.

Table 1: Cost of one iteration of RCPI OVA, ERCPI, and BRCPI. S is the number of states, A the number of actions, K the number of rollouts, T is trajectory length, $\mathcal{C}(S, A)$ is the cost of learning a classifier for S states, A actions.

Algorithm	Simulation Cost	Learning Cost
RCPI-OVA	$SAK(TA)$	$A.\mathcal{C}(S)$
ERCPI	$SAK(T\gamma\log(A))$	$\gamma\log(A).\mathcal{C}(S)$
BRCPI	$\gamma\log(A)\ (2SK(2T))$	$\gamma\log(A)\mathcal{C}(S)$

Table 2: Complexity w.r.t. the number of possible actions.

Method	RCPI OVA	ERCPI	BRCPI
Complexity	$\mathcal{O}(A^2)$	$\mathcal{O}(A\log(A))$	$\mathcal{O}(\log(A))$

When considering the BRCPI algorithm, $\mathcal{I}$ and $\mathcal{C}$ are reduced as in ERCPI. However, the simulation cost is reduced as well, as our method proposes to learn a set of optimal binary policies on $\gamma\log(A)$ binary sub-MDPs. For each of these sub-problems, the simulation cost is $2SK(2T)$ since the number of possible actions is only 2. The learning cost corresponds to learning only $\gamma\log(A)$ binary classifiers resulting in a very low cost — cf. Table 1. The overall resulting complexity w.r.t. to the number of actions is presented in Table 2, showing that the complexity of BRCPI is only logarithmic. In addition, it is important to note

[4] In practice, when there are many actions, simulation cost is significantly higher than learning cost, which is thus ignored [7].

[5] We do not consider the computational cost of transitions in the MDP.

that each of the BRCPI sub-problems is atomic, and are therefore easily parallelized. To illustrate these complexities, computation times are reported in the experimental section.

4 Experiments

In this paper, our concern is really about being able to deal with a large number of uncorrelated actions in practice. Hence, the best demonstration of this ability is to provide an experimental assessment of ERCPI and BRCPI. In this section, we show that BRCPI exhibits very important speed-ups, turning days of computations into hours or less.

4.1 Protocol

We evaluate our approaches on two baseline RL problems: Mountain Car and Maze.

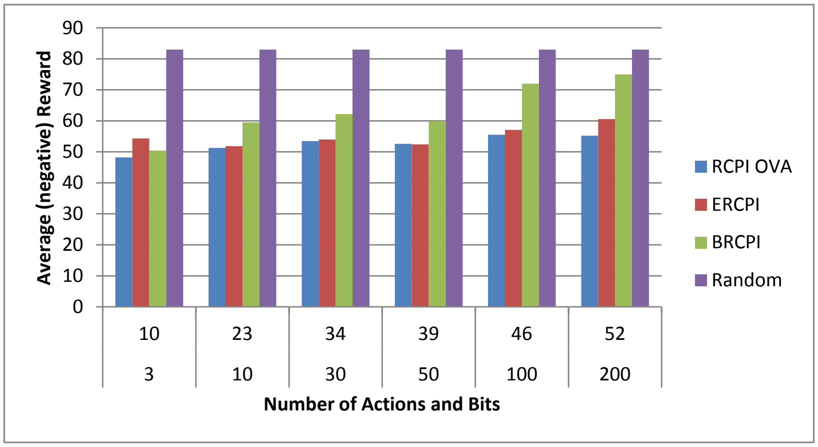

Fig. 2: **Mountain Car**: Average reward (negative value: the smaller, the better) obtained by the different algorithms on 3 runs with different numbers of actions. On the X-axis, the first line corresponds to $\gamma \log(A)$ while the second line is the number of actions A.

The first problem, **Mountain Car**, is well-known in the RL community. Its definition varies, but it is usually based on a discrete and small set of actions (2 or 3). However, the actions may be defined over a continuous domain, which is more "realistic". In our experiment, we discretize the range of accelerations to obtain a discrete set of actions. Discretization ranges from coarse to fine in the experiments, thus allowing us to study the effect of the size of the action set on the performance of our algorithms. The continuous *state* space is handled by way of tiling [8]. The reward at each step is -1, and each episode has a maximum length of 100 steps. The overall reward thus measures the ability of the obtained policy to push the car up to the mountain quickly.

The second problem, **Maze**, is a 50x50 grid-world problem in which the learner has to go from the left side to the right side of a grid. Each cell of the grid corresponds to a particular negative reward, either -1, -10, or -100. For the simplest case, the agent can choose either to move *up*, *down*, or *right*, resulting in a 3-action MDP. We construct more complex action sets by generating all sequences of actions of a defined length i.e. for length 2, the 6 possible actions are *up-up*, *up-right*, *down-up*, etc. Contrary to Mountain Car, there is no notion of similarity between actions in this maze problem w.r.t. their consequences. Each state is represented by a vector of features that contains the information about the different types of cells that are contained in a 5x5 grid around the agent. The overall reward obtained by the agent corresponds to its ability to go from the left to the right, avoiding cells with high negative rewards.

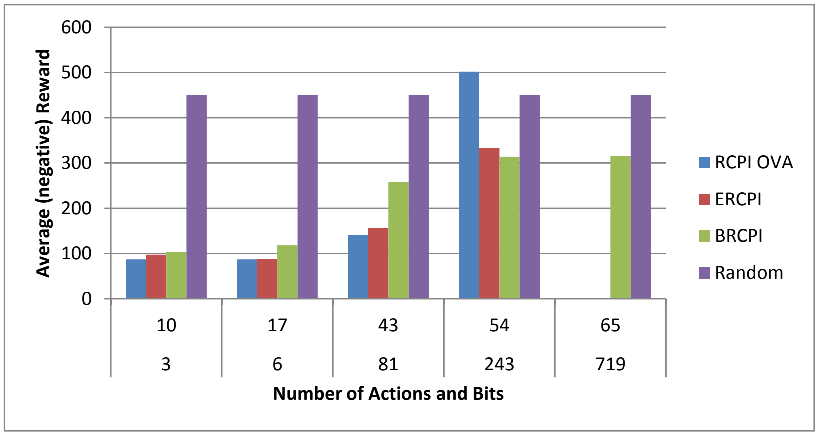

Fig. 3: **Maze**: Average reward (negative value: the smaller, the better) obtained by the different algorithms on 3 different random mazes with different numbers of actions. On the X-axis, the first line corresponds to $\gamma \log(A)$ while the second line is the number of actions A. OVA and ERCPI were intractable for 719 actions. Note that for 243 actions, RCPI-OVA learns a particularly bad policy.

In both problems, training and testing states are sampled uniformly in the space of the possible states. We have chosen to sample $S = 1000$ states for each problem, the number of trajectories made for each state-action pair is $K = 10$.

The binary base learner is a hinge-loss perceptron learned with 1000 iterations by stochastic gradient-descent algorithm. The error correcting codes have been generated using a classical random procedure as in [9]. The α-value of the alpha-mixture policy is 0.5.

4.2 Results

The average rewards obtained after convergence of the three algorithms are presented in Figures 3 and 2 with a varying number of actions. The average reward of a random policy is also illustrated. First of all, one can see that RCPI-OVA and ERCPI perform similarly on both problems except for Maze with 243 actions. This can be explained by the fact that OVA strategies are not able to deal with problems with many classes when they involve solving binary classification problems with few positive examples. In this setting, ECOC-classifiers are known to perform better. BRCPI achieves lower performances than OVA-RCPI and ERCPI. Indeed, BRCPI learns optimal independent binary policies that, when used together, only correspond to a sub-optimal overall policy. Note that even with a large number of actions, BRCPI is able to learn a relevant policy — in particular, Maze with 719 actions shows BRCPI is clearly better than the random baseline, while the other methods are simply intractable. This is a very interesting result since it implies that BRCPI is able to find non-trivial policies when classical approaches are intractable.

Table 3 provides the computation times for one iteration of the different algorithms for Mountain Car with 100 actions. ERCPI speeds-up RCPI by a factor 1.4 while BRCPI is 12.5 times faster than RCPI, and 23.5 times faster when considering only the simulation cost. This explains why Figure 3 does not show performances obtained by RCPI and ERCPI on the maze problem with 719 actions: in that setting, one iteration of these algorithms takes days while only requiring a few hours with BRCPI. Note that these speedup values increase with the number of actions.

At last, Figure 4 gives the performance of BRCPI depending on the number of rollouts, and shows that a better policy can be found by increasing the value of K. Note that, even if we use a large value of K, BRCPI's running time remains low w.r.t. to OVA-RCPI and ERCPI.

5 Related Work

Rollout Classification Policy Iteration [5] provides an algorithm for RL in MDPs that have a very large state space. RCPI's Monte-Carlo sampling phase can be very costly, and a couple approaches have been provided to better sample the state space [10], thus leading to speedups when using RCPI. Recently, the effectiveness of RCPI has been theoretically assessed [7]. The well known efficiency of this method for real-world problems and its inability to deal with many actions have motivated this work.

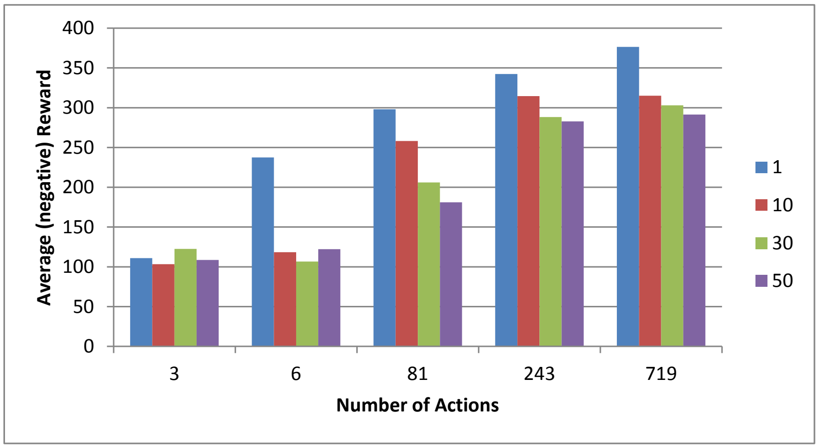

Fig. 4: **Maze Rollouts**: Average reward (negative value: the smaller, the better) obtained by BRCPI for $K = 1, 10, 30, 50$.

Reinforcement Learning has long been able to scale to state-spaces with many (if infinite) states by generalizing the value-function over the state space [11, 12]. Tesauro first introduced rollouts [13], leveraging Monte-Carlo sampling for exploring a large state and action space. Dealing with large action spaces has additionally been considered through sampling or gradient descent on Q [3, 1], but these approaches assume a well-behaved Q-function, which is hardly guaranteed.

There is one vein of work reducing action-space look-ups logarithmically by imposing some form of binary search over the action space [14, 15]. These approaches augment the MDP with a structured search over the action space, thus placing the action space's complexity in the state space. Although not inspirational to ERCPI, these approaches are similar in their philosophy. However, neither proposes a solution to speeding up the learning phase as BRCPI does, nor do they eschew value functions by relying solely on classifier-based approaches as ERCPI does.

Error-Correcting Output Codes were first introduced by Dietterich and Bakiri ([4]) for use in the case of multi-class classification. Although not touched upon in this article, coding dictionary construction can be a key element to the ability of the ECOC-based classifier's abilities[16]. Although in our case we rely on randomly generated codes, codes can be learned from the actual training data [17] or from an *a priori* metric upon the classes space or a hierarchy [18].

Table 3: Time (in seconds) spent for one iteration — during simulation and learning — of the different variants of the RCPI algorithms using a Xeon-X5690 Processor and a TESLA M2090 GPU for $K = 10$ and $S = 1000$. The total speedup (and simulation speedup) w.r.t. OVA-RCPI are presented on the last column.

Mountain Car - 100 Actions - 46 bits				
	Sim.	Learning	Total	Speedup
OVA	4,312	380	4,698	×1.0
ERCPI	3,188	190	3,378	×1.4(×1.35)
BRCPI	184	190	374	×12.5(×23.5)

6 Conclusion

We have proposed two new algorithms which aim at obtaining a good policy while learning faster than the standard RCPI algorithm. ERCPI is based on the use of Error Correcting Output Codes with RCPI, while BRCPI consists in decomposing the original MDP in a set of binary-MDPs which can be learned separately at a very low cost. While ERCPI obtains equivalent or better performances than the classical *One Vs. All* RCPI implementations at a lower computation cost, BRCPI allows one to obtain a sub-optimal policy very fast, even if the number of actions is very large. We believe that there are plenty of high-complexity situations where having a policy that is even slightly better than random can be very advantageous; in the case of ERCPI we can get sub-optimal policies rapidly, which provide at least *some* solution to an otherwise intractable problem. The complexity of the proposed solutions are $\mathcal{O}(A\log(A))$ and $\mathcal{O}(\log(A))$ respectively, in comparison to RCPI's complexity of $\mathcal{O}(A^2)$. Note that one can use BRCPI to discover a good policy, and then ERCPI in order to improve this policy; this practical solution is not studied in this paper.

This work opens many new research perspectives: first, as the performance of BRCPI directly depends on the quality of the codes generated for learning, it can be very interesting to design automatic methods able to find the well-adapted codes, particularly when one has a metric over the set of possible actions. From a theoretical point of view, we plan to study the relation between the performances of the sub-policies π_i in BRCPI and the performance of the final obtained policy π. At last, the fact that our method allows one to deal with problems with thousands of discrete actions also opens many applied perspectives, and can allow us to find good solutions for problems that have never been studied before because of their complexity.

Acknowledgements This work was partially supported by the French National Agency of Research (Lampada ANR-09-EMER-007). The authors gratefully acknowledge the many discussions with Moustapha Cisse.

Bibliography

[1] Lazaric, A., Restelli, M., Bonarini, A.: Reinforcement Learning in Continuous Action Spaces through Sequential Monte Carlo Methods. In: Proc. of NIPS '07. (2007)

[2] Bubeck, S., Munos, R., Stoltz, G., Szepesvári, C., et al.: X-armed bandits. Journal of Machine Learning Research **12** (2011) 1655–1695

[3] Negoescu, D., Frazier, P., Powell, W.: The knowledge-gradient algorithm for sequencing experiments in drug discovery. INFORMS J. on Computing **23**(3) (2011) 346–363

[4] Dietterich, T., Bakiri, G.: Solving multiclass learning problems via error-correcting output codes. Jo. of Art. Int. Research **2** (1995) 263–286

[5] Lagoudakis, M.G., Parr, R.: Reinforcement learning as classification: Leveraging modern classifiers. In: Proc. of ICML '03. (2003)

[6] Bentley, J.L.: Multidimensional binary search trees used for associative searching. Communications of the ACM **18**(9) (1975) 509–517

[7] Lazaric, A., Ghavamzadeh, M., Munos, R.: Analysis of a classification-based policy iteration algorithm. In: Proc. of ICML '10. (2010) 607–614

[8] Sutton, R.: Generalization in reinforcement learning: Successful examples using sparse coarse coding. In: Proc. of NIPS '96. (1996) 1038–1044

[9] Berger, A.: Error-correcting output coding for text classification. In: Workshop on Machine Learning for Information Filtering, IJCAI '99. (1999)

[10] Dimitrakakis, C., Lagoudakis, M.G.: Rollout sampling approximate policy iteration. Machine Learning **72**(3) (July 2008) 157–171

[11] Tham, C.: Modular on-line function approximation for scaling up reinforcement learning. PhD thesis, University of Cambridge (1994)

[12] Tesauro, G.: Practical issues in temporal difference learning. Machine Learning **8** (1992) 257–277

[13] Tesauro, G., Galperin, G.R.: On-Line Policy Improvement Using Monte-Carlo Search. In: Proc. of NIPS '97. (1997) 1068–1074

[14] Pazis, J., Lagoudakis, M.G.: Reinforcement Learning in Multidimensional Continuous Action Spaces. In: Proc. of Adaptive Dynamic Programming and Reinf. Learn. (2011) 97–104

[15] Pazis, J., Parr, R.: Generalized Value Functions for Large Action Sets. In: Proc. of ICML '11. (2011) 1185–1192

[16] Beygelzimer, A., Langford, J., Zadrozny, B.: Machine learning techniques—reductions between prediction quality metrics. Performance Modeling and Engineering (2008) 3–28

[17] Crammer, K., Singer, Y.: On the Learnability and Design of Output Codes for Multiclass Problems. Machine Learning **47**(2) (2002) 201–233

[18] Cissé, M., Artieres, T., Gallinari, P.: Learning efficient error correcting output codes for large hierarchical multi-class problems. In: Workshop on Large-Scale Hierarchical Classification ECML/PKDD '11. (2011) 37–49