Probabilistic Policy Reuse for Inter-Task Transfer Learning

Fernando Fernández [a] Javier García [a] Manuela Veloso [b]

[a] *Universidad Carlos III de Madrid*

[b] *Carnegie Mellon University*

Abstract

Policy Reuse is a reinforcement learning technique that efficiently learns a new policy by using past similar learned policies. The Policy Reuse learner improves its exploration by probabilistically including the exploitation of those past policies. Policy Reuse was introduced and previously demonstrated its effectiveness in problems with different reward functions in the same state and action spaces. In this article, we contribute Policy Reuse as transfer learning among different domains. We introduce extended MDPs to include domains and tasks, where domains have different state and action spaces, and task are problems with different rewards within a domain. We show how Policy Reuse can be applied among domains by defining and using a mapping between their state and action spaces. We use several domains, as versions of a simulated RoboCup Keepaway problem, where we show that Policy Reuse can be used as a mechanism of transfer learning significantly outperforming a basic policy learner.

Key words: Reinforcement Learning, Transfer Learning, Policy Reuse

1 Introduction

Reinforcement Learning (RL) [1] is a powerful control learning technique based on a trial and error process guided by reward signals received from the environment. Classical RL algorithms as Q-Learning [2] rely on an intense exploration of the action and state spaces. In the presence of large spaces in complex domains, learning an optimal policy typically requires an extensive interaction of the learning agent with the environment.

Email addresses: ffernand@inf.uc3m.es (Fernando Fernández), fjaviergp@gmail.com (Javier García), veloso@cs.cmu.edu (Manuela Veloso).

Although the cost (time, resources, etc.) of the learning process may be very high, successful results have been reached to solve complex tasks [3,4]. But many different efforts continue to investigate how to address the potential complexity of reinforcement learning. Several such efforts rely on the appealing idea of reusing the knowledge acquired in one learning process to solve other problems, including the transfer of value functions [5], the reuse of options [6], or the learning of hierarchical modules [7]. The cost of the guided learning is consistently reduced.

Policy Reuse is a technique where the learner is guided by past policies balancing among its search for the optimal policy among three choices: the exploitation of the ongoing learned policy, the exploration of new random actions, and the exploitation of past policies [8]. Policy Reuse builds upon two main contributions: (i) an exploration strategy able to probabilistically bias the exploration of the domain with a predefined past policy; (ii) a similarity metric that allows the estimation of the similarity of past policies with respect to a new one. Policy Reuse has been demonstrated in a series of complex grid-based learning tasks where the efficiency of the learner significantly improves when reusing past policies [8]. Although the grid-based tasks have been extensively used in the evaluation of reinforcement learning for robot control, there remained the question of how Policy Reuse could be applied to domains potentially more complex, such as the Keepaway domain [4]. The challenge is to transfer learned knowledge from a simple to a larger state and action spaces, e.g., from a keepaway problem with some number of teammates and opponents to a new one with larger number of agents. Our work is motivated by this domain and contributes a method to apply Policy Reuse to transfer learning.

Policy Reuse needs that the past policies and the policy to be learned be defined in the same state and action spaces. That constraint is required to allow the agent to execute policies previously learned in the current task. In this paper, we introduce an extension of Policy Reuse that allows to reuse past policies even when they are defined in different state and action spaces. The method is based on a mapping among policies. Such mappings between state and action spaces are required in other approaches for transfer learning [9,10]. Given that we only need to transfer policies for policy reuse, interestingly and as shown in this paper, the amount of knowledge required by our mapping is much smaller than in methods based on the transfer of the value function. In our case, the mapping is assumed to be given. Some methods try to solve this problem through a study of actions correlations [11], through state abstraction [12], or by defining the relationships between the state variables of the source and target MDP's [13].

We demonstrate empirically that Policy Reuse, combined with the action and state space mapping, is able to improve the learning performance if compared

with learning with no reuse. We apply the transfer learning using Policy Reuse to the Keepaway framework [4], which favorably compares with other transfer learning methods that have been recently applied [14,10,15] to this same domain.

2 Policy Reuse

We summarize Policy Reuse. Firstly, we describe the concepts of task, domain, and gain. Then, we define how the reuse of a past policy is used as a probabilistic bias in a new exploratory process. We also briefly introduce a similarity concept between policies. Lastly, we review the PRQ-Learning algorithm [8].

2.1 Domains, Tasks and MDPs

A Markov Decision Process is a tuple $< S, A, T, R >$, where S is the set of all possible states, A is the set of all possible actions, T is a stochastic state transition function, $T : S \times A \times S \rightarrow \Re$, and R is a stochastic reward function, $R : S \times A \rightarrow \Re$ [1]. Reinforcement learning (RL) assumes that T and R are unknown. We focus on RL domains where different **tasks** can be solved. We introduce a task as a specific reward function, while S, A, and T stay constant for all the tasks. Thus, we extend the concept of an MDP by introducing two new concepts: domain and task. We characterize a domain, D, as a tuple $< S, A, T >$. We define a task, Ω, as a tuple $< D, R_\Omega >$, where D is a domain as defined before, and R_Ω is the stochastic and unknown reward function.

Learning proceeds in multiple **episodes**. Each episode positions the learning agent in an initial state of the environment and terminates when an end condition is satisfied, for instance, when a maximum number of steps, say H, is achieved. Thus, the learning objective is to maximize the expected average reinforcement per episode, say W, defined as $W = \frac{1}{K} \sum_{k=0}^{K} \sum_{h=0}^{H} \gamma^h r_{k,h}$, where K is the total number of episodes, $r_{k,h}$ is the immediate reward obtained in the step h of the episode k, and γ ($0 \leq \gamma \leq 1$) discounts future rewards. An action policy, $\Pi : S \rightarrow A$, defines for each state, the action to execute. The action policy Π^* is optimal if it maximizes the gain W in the task Ω, W^*_Ω.

Policy Reuse aims at speeding up the learning process of a new task by using past policies that solve different similar tasks to bias the exploration process. Then, Policy Reuse with the objective of solving a task Ω, i.e., to learn Π^*_Ω faces two main steps: (i) to previously solve a set of tasks $\{\Omega_1, \ldots, \Omega_n\}$ resulting in a set of policies, $\{\Pi^*_1, \ldots, \Pi^*_n\}$; (ii) to use the policies, Π^*_i to learn the new

policy Π^*_Ω.

An efficient solution to Policy Reuse is the PRQ-Learning algorithm [8], which automatically answers two questions: (i) which policy, from the set $\{\Pi^*_1, \ldots, \Pi^*_n\}$, should be used to bias the new learning process, and (ii) once a policy Π_i is selected, how is it integrated in the learning process. PRQ-Learning is based on an exploration strategy, π-reuse, that is able to bias the learning of a new policy with one past policy. Such strategy enables the identification of a similarity metric between policies, providing a method to select the most accurate policy to reuse. Both the π-reuse strategy and the similarity metric, defined in [8], are now summarized.

2.2 A Similarity Metric Between Policies

The goal of the π-reuse strategy is to balance random exploration, exploitation of the past policy, and exploitation of the new policy being learned. The π-reuse strategy follows the past policy, Π_{past} and the new policy with a probability ψ and $1 - \psi$, respectively. As random exploration is always required, when exploiting the new policy, π-reuse follows an ε-greedy strategy, as defined in Table 1. The υ parameter decays the value of ψ in each episode.

π-reuse ($\Pi_{past}, K, H, \psi, \upsilon$).

Initialize $Q^{\Pi_{new}}(s, a) = 0$, $\forall s \in S, a \in A$
For $k = 1$ to K
  Set the initial state, s, randomly.
  Set $\psi_1 \leftarrow \psi$
  for $h = 1$ to H
    With a probability of ψ_h, $a = \Pi_{past}(s)$
    With a probability of $1 - \psi_h$, $a = \varepsilon$-greedy($\Pi_{new}(s)$)
    Receive current state s', and reward, $r_{k,h}$
    Update $Q^{\Pi_{new}}(s, a)$, and therefore, Π_{new}:

$$Q^{\Pi_{new}}(s, a) \leftarrow (1 - \alpha) Q(s, a)^{\Pi_{new}} + \alpha [r + \gamma \max_{a'} Q^{\Pi_{new}}(s', a')]$$

    Set $\psi_{h+1} \leftarrow \psi_h \upsilon$
    Set $s \leftarrow s'$
$W = \frac{1}{K} \sum_{k=0}^{K} \sum_{h=0}^{H} \gamma^h r_{k,h}$
Return W, $Q^{\Pi_{new}}(s, a)$ and Π_{new}

Table 1
π-reuse Exploration Strategy.

Interestingly, the π-reuse strategy also contributes a similarity metric between policies, based on the gain obtained when reusing each policy. Let W_i be the gain obtained while executing the π-reuse exploration strategy, reusing the past policy Π_i.

W_i is used as an estimation of how similar the policy Π_i is to the one we are currently learning. The set of W_i values, for $i = 1, \ldots, n$, is unknown a priori, but it can be estimated on-line while the new policy is computed in the

different episodes. This idea is formalized in the PRQ-Learning algorithm.

2.3 PRQ-Learning Algorithm

Table 2 presents the PRQ-Learning algorithm (Policy Reuse in Q-Learning) [8]. The learning algorithm used is Q-Learning [2]. The goal is to solve a task Ω, i.e., to learn an action policy Π_Ω. We have a Policy Library $L = \{\Pi_1, \ldots, \Pi_n\}$ composed of n past policies. Then, W_i is the expected average reward that is received when reusing the policy Π_i with the π-reuse exploration strategy, and W_Ω is the average reward that is received when following the policy Π_Ω greedily. The algorithm uses the W values in a softmax way to choose between reusing a past policy with the π-reuse exploration strategy, or following the ongoing learned policy greedily.

PRQ-Learning(Ω, L, K, H)

- Given:

(1) A new task Ω we want to solve
(2) A Policy Library $L = \{\Pi_1, \ldots, \Pi_n\}$
(3) A maximum number of episodes to execute, K
(4) A maximum number of steps per episode, H

- Initialize:

(1) $Q_\Omega(s, a) = 0, \forall s \in S, a \in A$
(2) $W_\Omega = W_i = 0$, for $i = 1, \ldots, n$

- For $k = 1$ to K do
  - Choose an action policy, Π_k, assigning to each policy the probability of being selected computed by the following equation:

$$P(\Pi_j) = \frac{e^{\tau W_j}}{\sum_{p=0}^{n} e^{\tau W_p}}$$

    where W_0 is set to W_Ω
  - Execute the learning episode k
    If $\Pi_k = \Pi_\Omega$, execute a Q-Learning episode following a fully greedy strategy
    Otherwise, call π-reuse($\Pi_k, \ldots, H, \psi, u$)
    In any case, receive the reward obtained in that episode, say R, and the updated Q function, $Q_\Omega(s, a)$
  - Recompute W_k using R

- Return the policy derived from $Q_\Omega(s, a)$

Table 2
PRQ-Learning.

The PRQ-algorithm has demonstrated to successfully and effectively reuse a predefined set of policies. In addition, algorithms to build the Policy Library have been contributed [8].

Up to this point, Policy Reuse has shown to require that the action and state spaces of the different policies, and therefore, the transition function, are homogeneous. However, a core contribution of this work consists on demonstrat-

ing that Policy Reuse can also be applied among policies learned in different state and action spaces, by creating a mapping between them.

2.4 Policy Reuse Across Tasks with Different State and Action Spaces

We assume a past policy, Π_{past} that solves the task $\Omega_{past} =< D_{past}, R_{past} >$, where $D_{past} =< S_{past}, A_{past}, T_{past} >$. We want to learn a new policy Π_{new} to solve a new task $\Omega_{new} =< D_{new}, R_{new} >$, where $D_{new} =< S_{new}, A_{new} >$. As a transfer learning goal, we want to reuse the past policy, Π_{past} to learn the new problem Π_{new}. Interestingly, given that our policy reuse method relies on policies, we only need to map states and actions, and we do not need to make any mapping between tasks, rewards nor transition functions. Thus, we need to find a mapping ρ that, given the policy Π_{past} in the domain D_{past}, outputs a new policy $\hat{\Pi}_{past}$ that is executable in the domain D_{new}. Following the ideas introduced in [9] Equation 1 defines the mapping $\rho =< \rho_S, \rho_A >$, where $\rho_A : A_{past} \rightarrow A_{new}$ is a function that maps the actions in the space A_{past} to the actions in the space A_{new}, and $\rho_S : S_{new} \rightarrow S_{past}$ maps states in the space S_{new} to the space S_{past}.

$$\hat{\Pi}_{past}(s) = \rho_A(\Pi_{past}(\rho_S(s))) \quad (1)$$

The way to make this mapping, i.e., to define ρ_S and ρ_A, depends on the source and the target tasks. For instance, this mapping can be done by an expert who decides the correspondence among states and actions [14,10]. That mapping can also be learned by observing a mentor [16]. The advantage of Policy Reuse over value function based transfers is that Policy Reuse only requires the mapping between the different state and action spaces, while value function based transfer requires also the mapping among such value function, which must be defined ad-hoc depending on the function approximator used [14]. Next section describes the application of Policy Reuse for transfer learning in Keepaway. The mapping among the different state and action spaces is based on knowledge inserted by the expert, and independent of the function approximator used.

3 The Keepaway

Keepaway is a research framework defined in the robot soccer simulation [4]. It consists of two teams, the keepers and the takers. The behavior of the takers is fixed and defined a priori and consists of several low level skills as "Go to Ball" or "Block a Pass," that are combined following some predefined rules.

There are also some low level skills defined for the keepers, as "Go to Ball" or "Hold the Ball." A task consists of learning a high level control function of those skills for each of the keepers. The goal is to maximize the time that the keepers maintain the possesion of the ball. The end condition for each episode is that the keepers lose the ball or that the ball goes out of a fixed sub-area of the field. Both conditions are also predefined.

We follow the framework defined in [14]. Thus, each keeper learns its own behavior, so we assume that each of them is implemented as a reinforcement learning agent, as explained next.

The state features and available actions use information derived from distances and angles between the keepers, the takers, and the center of the play area. Depending on the number of keepers and takers, the features used are different. Table 3 shows some features of the state space when 3 keepers play against 2 takers (3vs2-keepaway), when 4 keepers play against 3 takers (4vs3-keepaway), and when 5 keepers play against 4 takers (5vs4-keepaway) [9]. Keepers and takers are ordered taking into account their respective distance to the ball, so the assignment of numbers to the keepers and the takers may change at each step [4].

3vs2-keepaway	4vs3-keepaway	5vs4-keepaway
$dist(k_1, C)$	$dist(k_1, C)$	$dist(k_1, C)$
$dist(k_2, C)$	$dist(k_2, C)$	$dist(k_2, C)$
$dist(k_3, C)$	$dist(k_3, C)$	$dist(k_3, C)$
	$dist(k_4, C)$	$dist(k_4, C)$
		$dist(k_5, C)$
...	...	...
$\min(dist(k_3, t_1), dist(k_3, t_2))$	$\min(dist(k_3, t_1), dist(k_3, t_2), dist(k_3, t_3))$ $\min(dist(k_4, t_1), dist(k_4, t_2), dist(k_4, t_3))$	$\min(dist(k_3, t_1), dist(k_3, t_2), dist(k_3, t_3), dist(k_3, t_4))$ $\min(dist(k_4, t_1), dist(k_4, t_2), dist(k_4, t_3), dist(k_4, t_3))$ $\min(dist(k_5, t_1), dist(k_5, t_2), dist(k_5, t_3), dist(k_5, t_3))$
13 features	19 features	25 features

Table 3
State spaces of the different keepaway versions.

The number of continuous features for the different versions of Keepaway are 13, 19 and 25 respectively. Thus, a method for state space generalization is required in order to improve the learning capabilities. Some methods for

3vs2-keepaway	4vs3-keepaway	5vs4-keepaway
Hold	Hold	Hold
Pass(k_2)	Pass(k_2)	Pass(k_2)
Pass(k_3)	Pass(k_3)	Pass(k_3)
	Pass(k_4)	Pass(k_4)
		Pass(k_5)

Table 4
Action spaces of the different keepaway versions.

function approximation, as CMAC or VQQL, have been applied to the Keepaway [5,19]. In this work, we use CMAC. Specifically, we have followed the approximation used in [4]. The only difference of our CMAC implementation in the Keepaway is that we use a separate value function approximator for each discrete action, following the ideas introduced in [17]. The complete details of the implementation can be found at [18].

The action space is limited to the execution of two different behaviors, HoldBall() and PassBall(k_i). PassBall receives a parameter, which is the player who will receive the pass. Table 4 defines the complete set of actions for the different number of keepers and takers.

The number of actions also grows with the number of keepers. The keeper that is holding the ball is the only one that makes a decision among the actions shown in Table 4. If the player does not hold the ball, it executes a predefined behavior as well as the takers do.

Last, the reward is the time that the team holds the ball since the agent executed the last action until the ball returns to it. Therefore, the reward does not depend only on the executed action, but also on the actions executed by the other agents.

4 Evaluation

This section describes the evaluation of the Policy Reuse approach for transfer learning in Keepaway. Additional evaluation of Policy Reuse in a robot navigation domain can be found in [8].

4.1 Tasks in the Keepaway Framework

We differentiate three different tasks, each of them defined in a different domain (different state and action spaces): (i) the 3vs2-keepaway, where 3 keepers play against two players; (ii) the 4vs3-keepaway; and (iii) the 5vs4-keepaway,

where 5 keepers play against 4 takers. The three tasks are learned in the order that have been defined. We consider two learning scenarios. In the first one, the three tasks are learned from scratch, so the knowledge acquired in one of them is never used to improve the learning process of the others. In the second case, we first learn the 3vs2-keepaway. Then, the agents that participated in 3vs2-keepaway, reuse the acquired policy to solve the 4vs3-keepaway. The agent that did not participate in the 3vs2-keepaway learns from scratch. Last, when learning the 5vs4-keepaway, each agent reuses the previously learned polices (2 policies for keepers 1, 2, and 3; 1 for keeper 4, and none for keeper 5), as defined in Table 5.

	3vs2-keepaway	4vs3-keepaway	5vs4-keepaway
Keeper 1	Learn $\Pi^{k_1}_{3vs2}$ from scratch	Learn $\Pi^{k_1}_{4vs3}$ by reusing $L^{k_1} = \{\Pi^{k_1}_{3vs2}\}$	Learn $\Pi^{k_1}_{5vs4}$ by reusing $L^{k_1} = \{\Pi^{k_1}_{3vs2}, \Pi^{k_1}_{4vs3}\}$
Keeper 2	Learn $\Pi^{k_2}_{3vs2}$ from scratch	Learn $\Pi^{k_2}_{4vs3}$ by reusing $L^{k_2} = \{\Pi^{k_2}_{3vs2}\}$	Learn $\Pi^{k_2}_{5vs4}$ by reusing $L^{k_2} = \{\Pi^{k_2}_{3vs2}, \Pi^{k_2}_{4vs3}\}$
Keeper 3	Learn $\Pi^{k_3}_{3vs2}$ from scratch	Learn $\Pi^{k_3}_{4vs3}$ by reusing $L^{k_3} = \{\Pi^{k_3}_{3vs2}\}$	Learn $\Pi^{k_3}_{5vs4}$ by reusing $L^{k_3} = \{\Pi^{k_3}_{3vs2}, \Pi^{k_3}_{4vs3}\}$
Keeper 4	Not Playing	Learn $\Pi^{k_4}_{4vs3}$ from scratch	Learn $\Pi^{k_4}_{5vs4}$ by reusing $L^{k_4} = \{\Pi^{k_4}_{4vs3}\}$
Keeper 5	Not Playing	Not Playing	Learn $\Pi^{k_5}_{5vs4}$ from scratch

Table 5
Description of the tasks solved.

The next section formalizes Policy Reuse in this scope. It also describes how each agent performs the mapping between the 3vs2-keepaway and the 4vs3-keepaway. Mappings from 3vs2-keepaway to 5vs-4 keepaway or from 4vs3-keepaway to 5vs-4 keepaway are equivalent, and easily instantiated from this case.

4.2 Policy Reuse in the Keepaway

The 3vs2-keepaway is a task, $\Omega^{3vs2} =< D^{3vs2}, R >$, defined in the domain D^{3vs2}, with a reward function R. The domain is defined as a tuple, $D^{3vs2} =< S^{3vs2}, A^{3vs2}, T^{3vs2} >$. S^{3vs2} and A^{3vs2} were defined in tables 3 and 4 respectively. Both the transition function T^{3vs2} and the reward function are unknown for the agent. The goal is to learn an action policy $\Pi_{\Omega^{3vs2}} : S^{3vs2} \rightarrow A^{3vs2}$ that outputs actions, given any state of the discretized state space. In a similar way, the 4vs3-keepaway task is formalized as following: $D^{4vs3} =< S^{4vs3}, A^{4vs3}, T^{4vs3} >$; $\Omega^{4vs3} =< D^{4vs3}, R >$; and $\Pi_{\Omega^{4vs3}} : S^{4vs3} \rightarrow A^{4vs3}$.

Following the notation introduced in Section 2.4, the mapping from a policy in the 3vs2-keepaway to the 4vs3-keepaway is performed as defined in equation 2:

$$\hat{\Pi}_{\Omega^{4vs3}}(s) = \rho_A(\Pi_{\Omega^{3vs2}}(\rho_S(s))) \quad (2)$$

where ρ_S is a function $\rho_S : S^{4vs3} \rightarrow S^{3vs2}$ that, given a state in the 4vs3-keepaway state space, S^{4vs3}, projects it on the 3vs2-keepaway state space, S^{3vs2}; and ρ_A is a function $\rho_A : A^{3vs2} \rightarrow A^{4vs3}$ that, given an action in the 3vs2-keepaway action space, A^{3vs2}, maps it on the 4vs3-keepaway action space.

In Keepaway, these projections are derived from the semantic of the features and actions [9]. In the case of the action spaces, $\rho_A(a) = a$, i.e. is the identity function, given that the action space in 3vs2-Keepaway is a subspace of the 4vs3-keepaway one. In the case of ρ_S, the mapping is a projection of the 4vs3-keepaway state space into the 3vs2-keepaway state space. This projection is derived from Table 3. Each feature in 4vs3-keepaway maps to the feature of 3vs2-keepaway in the same row. For instance, feature $\min(dist(k_2,t_1),dist(k_2,t_2),dist(k_2,t_3))$ in 4vs3-keepaway maps to the feature $\min(dist(k_2,t_1),dist(k_2,t_2))$ in 3vs2-keepaway. The features in 4vs3-keepaway that does not have equivalent in the 3vs2-keepaway are eliminated, as it is the case of the feature $\min(dist(k_4,t_1),dist(k_4,t_2),dist(k_4,t_3))$. The reason is that it only involves information about the keeper $k4$, that does not play in 3vs2-keepaway.

4.3 Parameter Setting

In the experiments, we have used the Keepaway layer Framework 0.6 [1]. We used the same settings as a previous transfer learning paper that uses this domain as a testbed [14], namely the field size is 25×25, vision capabilities are set to full, and the synchronous mode is set on to speed up the simulator.

In order to have some baseline results, in all the cases we introduce the performance of a policy where the agents always passes to the second keeper (which outperforms the policy where the agents always hold the ball and the random policy, used for comparisons in other papers [4]).

In our case, we use Sarsa(λ) for approximating the optimal Q-function. The goal of using this algorithm is to show that Policy Reuse can be applied directly with other Reinforcement Learning algorithms, including on-policy methods: generalize the PRQ-Learning algorithm defined in Section 2.3 only requires

[1] The software is available in the Keepaway Player Framework web page: http://www.cs.utexas.edu/users/AustinVilla/sim/keepaway/

to change the Q update equations and the way such updates are performed. A second objective is that the results can be compared directly with previous transfer learning approaches applied in the Keepaway domain that used Sarsa. Additional evaluations of Policy Reuse in the Keepaway with different TD methods (Q-Learning or Q(λ)) and different function approximation approaches (CMAC and VQQL) can be found in previous works [19,18].

The parameter setting is the following: In the Sarsa(λ), $\alpha = 0.125$, $\gamma = 1$ and $\lambda = 0.5$, as defined in other previous works [14]; in the π-reuse exploration strategy, $\psi = 1$, $\upsilon = 0.95$ and $\varrho = 1 - \psi_h$; and in the PRQ-Learning algorithm, τ is initialized to 0, and incremented by 0.05 in each episode. All these parameters have taken the same values that in the previous experimentation in a navigation domain [8], which demonstrated to be accurate for that task. Thus, they may not be optimal for Keepaway, but provide us with results that are accurate enough for our study on Policy Reuse.

When learning from scratch, an ϱ-greedy strategy is followed, increasing the value of ϱ from 0 (random behavior) to 1 (greedy behavior) by 0.0001 in each episode. As before, this strategy demonstrated to provide good results in this domain, but no extensive parameter tuning was performed.

4.4 Results

4.4.1 3vs2-keepaway

Firstly, we have learned the 3vs2-keepaway. The results are summarized in Figure 1. The x axis of the figure describes the training time, while the y axis shows the episode duration, which is the value to maximize. In this task, the random behavior obtains a performance of 7.25 seconds, while the policy "Pass K2" obtains an average value of 8.17. When learning, the average values raises from 7.25 up to around 25 seconds in the two different executions performed.

4.4.2 4vs3-keepaway

After learning the 3vs2-keepaway, the agents are given the 4vs3-keepaway task. In this case, we followed two different approaches for learning: learning from scratch, and Policy Reuse, following the scheme introduced in Table 5. The results are summarized in Figure 2. In this task, the random behavior obtains a performance of 5.56 seconds, while the policy "Pass K2" obtains an average value of 5.83. These results are lower than the ones obtained in the 3vs2-keepaway, demonstrating that this task is considerably harder. Again, each learning process was performed twice to show that there are not significant differences between different executions.

Fig. 1. Results of learning in the 3vs2-keepaway

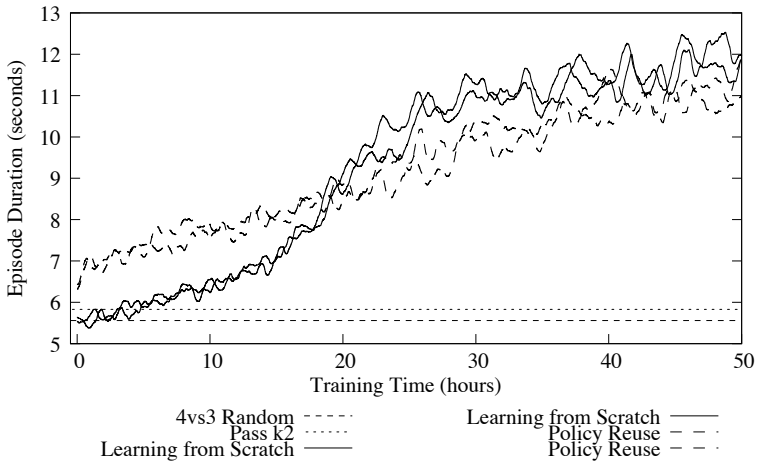

Fig. 2. Results of learning in the 4vs3-keepaway

When learning, we can differentiate two phases. The first one ranges from training time 0 until training time 20. In that phase, when learning from scratch, the performance grows from the same result than random behavior (around 5.5) up to almost 9 seconds. Policy Reuse, however, achieves the seven seconds almost from the early beginning, demonstrating the reusing the past policies allows to generate better behavior from the very first steps. During the initial hours, policy reuse outperforms leaning from scratch in around one second. After the 20 training hours, Policy Reuse has achieved also the 9 seconds. From the 20 training hours, learning from scratch seems to behave slightly better. After 50 training hours, both methods obtain results between 11 and 12 seconds.

This result is qualitatively different from the one reported on value function based behavior transfer [14]. In that work, both learning from scratch and behavior transfer obtain similar results in the initial steps of learning. The contribution of behavior transfer is that the accurate behaviors are achieved faster than when learning from scratch. However, in our case, the performance

of the task is much better from the early episodes of the learning. Another difference between our work and that one is that, given that they transfer the Q function, for instance, from 3vs2-keepaway to 4vs3-keepaway, they need to initialize some values of the Q-function by hand. For instance, the Q values of the action "Pass k3" in the 3vs2-keepaway are transfered both to the action "Pass k3" and to the action "Pass k4" of 4vs3-keepaway. This initialization of the Q values is required to make the transfered Q table more homogeneous than if some values are set to 0. Policy Reuse utilizes past policies only as a bias in the exploration, so we do not suffer that problem, and the amount of knowledge required for the transfer is lower.

Quantitatively, the results provided in [14] are similar, since the authors report that, after 20 training hours, they obtain values of 9 seconds for the performance, both when learning from scratch and when transferring behaviors. Those values are similar to the ones obtained in our work after 20 seconds. Similar conclusions can be obtained if we compare the results of Policy Reuse with the results reported in [13].

4.4.3 5vs4-keepaway

After learning the 3vs2 and 4vs3-keepaway, the situation of the agents, as defined in Table 5 is the following: the first three agents, that participated also in the 3vs2 and the 4vs3-keepaway, can reuse two action policies, learned in each task respectively; the fourth agent can reuse only the policy of the 4vs3-keepaway task, given that it did not participate in the 3vs2-keepaway; the fifth keeper never played before, so it needs to learn from scratch. The results are shown in Figure 3.

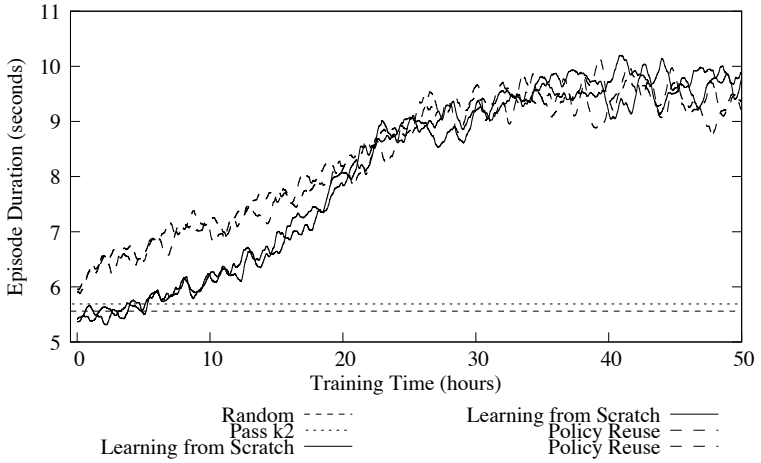

Fig. 3. Results of learning in the 5vs4-keepaway

When learning from scratch, the values raise from around 5.5 to almost 8 in 20 training hours. When reusing the past policies, the initial values are already close to 7, improving the performance of the initial steps of the learning. After

20 hours of learning, the differences between learning from scratch and by reuse are not significant, and are between 9 and 10 in both cases. Therefore, qualitatively, the results are similar to the 4vs3-keepaway. Some additional evaluations of Policy Results with different TD methods and functiona approximation methods does not show significant differences with the results reported here [18].

5 Conclusions

Policy Reuse is a powerful technique to improve learning performance in new robotic tasks by reusing policies learned in previous tasks. Policy Reuse does not assume that the previour knowledge is useful, but it reuses the past knowledge depending on whether it contributes to the exploration process or not. That utility, called reuse gain, is also discovered during the learning process.

A main contribution of Policy Reuse is that the learning robot executes good behaviors from the early episodes of learning. In the experiments shown in Keepaway, we demonstrate that Policy Reuse obtains episode durations of more than one second that learning from scratch in initial episodes, and that learning from scratch is only able to obtain similar results when it has being learning for more than 20 hours.

In this paper, we demonstrate two main issues. Firstly, that Policy Reuse, together with a function approximation method, is applicable in domains with a large state space. And second, that policies that solve tasks in different state and action spaces can be succesfully reused to learn policies in a different state/action space: it only requires a mapping between the different spaces, so past policies can be executed in the new spaces.

Acknowledgements

This research was conducted while the first author was visiting Carnegie Mellon from the Universidad Carlos III de Madrid, supported by a generous grant from the Spanish Ministry of Education and Fullbright. The authors have been partially sponsored also by the spanish Ministerio de Ciencia en Innovacin project number TIN2008-06701-C03-03 and by Comunidad de Madrid-UC3M project number CCG08-UC3M/TIC-4141.

References

[1] L. P. Kaelbling, M. L. Littman, A. W. Moore, Reinforcement learning: A survey, International Journal of Artificial Intelligence Research 4 (1996) 237–285.

[2] C. Watkins, Learning from delayed rewards, Ph.D. thesis, Cambridge University, Cambridge, England (1989).

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

[4] P. Stone, R. S. Sutton, G. Kuhlmann, Reinforcement learning for RoboCup-soccer keepaway, Adaptive Behavior 13 (3).

[5] M. E. Taylor, P. Stone, Y. Liu, Value functions for RL-based behavior transfer: A comparative study, in: Proceedings of the Twentieth National Conference on Artificial Intelligence, 2005.

[6] R. S. Sutton, D. Precup, S. Singh, Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning, Artificial Intelligence 112 (1999) 181–211.

[7] T. G. Dietterich, Hierarchical reinforcement learning with the MAXQ value function decomposition, Journal of Artificial Intelligence Research 13 (2000) 227–303.

[8] F. Fernández, M. Veloso, Probabilistic policy reuse in a reinforcement learning agent, in: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06), 2006.

[9] M. Taylor, P. Stone, Y. Liu, Transfer learning via inter-task mappings for temporal difference learning, Journal of Machine Learning Research 8 (1) (2007) 2125–2167.

[10] L. Torrey, T. Walker, J. Shavlik, R. Maclin, Using advice to transfer knowledge acquired in one reinforcement learning task to another, in: Proceedings of the European Conference on Machine Learning (ECML'05), 2005.

[11] M. E. Taylor, P. Stone, Inter-task action correlation for reinforcement learning tasks, in: Proceedings of the Twenty-first National Conference on Artificial Intelligence (AAAI'06), 2006.

[12] T. J. Walsh, L. Li, M. Littman, Transferring state abstractions between mdps, in: Proceedings of the ICML'06 Workshop on Structural Knowledge Tranfer for Machine Learning, 2006.

[13] V. Soni, S. Singh, Using homomorphisms to transfer options across continuous reinforcement learning domains, in: Proceedings of AAAI'06, 2006.

[14] M. E. Taylor, P. Stone, Behavior transfer for value-function-based reinforcement learning, in: The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, 2005.

[15] W. H. Hsu, S. J. Harmon, E. Rodríguez, C. Zhong, Empirical comparison of incremental reuse strategies in genetic programming for keepaway soccer, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'04), 2004.

[16] B. Price, C. Boutilier, Imitation and reinforcement learning in agents with heterogeneous actions, in: AI '01: Proceedings of the 14th Biennial Conference of the Canadian Society on Computational Studies of Intelligence, Springer-Verlag, London, UK, 2001, pp. 111–120.

[17] F. Fernández, D. Borrajo, Two steps reinforcement learning, International Journal of Intelligent Systems 23 (2) (2008) 213–245.

[18] F. J. García, M. Veloso, F. Fernández, Reinforcement learning in the robocup-soccer keepaway, in: Proceedings of the 12th Conference of the Spanish Association for Artificial Intelligence (CAEPIA'07+TTIA), 2007.

[19] F. Fernández, M. Veloso, Policy reuse for transfer learning across tasks with differentstate and action spaces, in: ICML'06 Workshop on Structural Knowledge Transfer for Machine Learning, 2006.