Policy Reuse for Transfer Learning Across Tasks with Different State and Action Spaces

Fernando Fernández FFERNAND@INF.UC3M.ES
Universidad Carlos III de Madrid, Avda. de la Universidad 30, 28911 Leganés, Madrid, Spain

Manuela Veloso VELOSO@CS.CMU.EDU
Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, USA

Abstract

Policy Reuse is a method for transfer learning in Reinforcement Learning in which learned policies are saved and reused in similar tasks. It relies on using the past policies as a probabilistic bias where the learning agent faces three choices: the exploitation of the ongoing learned policy, the exploration of random unexplored actions, and the exploitation of past policies. However, Policy Reuse requires that the state and action spaces of the reused policies are the same that the state and action spaces of the policy that the agent is currently learning. In this paper we demonstrate that to apply Policy Reuse for transfer learning among tasks with different state/action spaces only requires that the learning agent maps the states and actions in the past task with the states and actions of the new one. Then, the past policies can bias the new learning process following the Policy Reuse framework. We provide a formal definition of this mapping and we demonstrate in the Keepaway domain that Policy Reuse between different tasks can outperform the results of learning from scratch.

1. Introduction

Reinforcement Learning (RL) (Kaelbling et al., 1996; Sutton & Barto, 1998) is a powerful technique to learn to solve different kind of tasks. The learning process is based on a trial and error process guided by reward signals received from the environment. Classical RL algorithms as Q-Learning (Watkins, 1989) rely on an intensive exploration of the action and state spaces. Due to the "curse of dimensionality" of such spaces in complex domains, solving a task typically requires and extensive interaction of the learning agent with the environment.

Although the cost (time, resources, etc.) of such a learning process may be very high, sometimes the task can be tackled and successfully solved (Tesauro, 1992; Stone et al., 2005). There have been many different efforts to address the complexity of the learning. To reuse the knowledge acquired in the current learning process when solving future problems, so the cost of future learning processes is reduced, is an appealing idea. In RL, several efforts have been done in this line, like the transfer of the value functions (Taylor et al., 2005), the reuse of options (Sutton et al., 1999) and the learning of hierarchies which modules can be used in the future (Dietterich, 2000).

Policy Reuse is a Reinforcement Learning technique guided by past policies offering the challenge of balancing among three choices: the exploitation of the ongoing learned policy, the exploration of new random actions, and the exploitation of past policies (?). Policy Reuse is based on two main cornerstones. On the one hand, an exploration strategy able to probabilistically bias the exploration of the domain with a predefined past policy; on the other hand, a similarity metric that allows the estimation of the similarity of past policies with respect to a new one.

However, Policy Reuse requires that the past policies and the policy that needs to be learned are defined in the same state and action spaces. That constraint is required to allow the agent to execute in the current task policies previously learned. In this paper, we propose an extension of Policy Reuse that allows to reuse past policies even when they are defined in a differ-

Appearing in *Proceedings of the 23rd International Conference on Machine Learning*, Pittsburgh, PA, 2006.

ent state and action space. The method is based on a mapping among policies. This mapping is required in most of the approaches for transfer learning (Taylor & Stone, 2005; Torrey et al., 2005). However, given we only need to transfer policies, the amount of knowledge required by the mapping is much smaller than in methods based on the transfer of the value function. In our case, the mapping is given by expert knowledge, although some works demonstrate that such a mapping can also be learned (Price & Boutilier, 2001).

We demonstrate empirically that Policy Reuse, together with action and state space mapping, is able to improve the results of learning from scratch a set of tasks. The testbed used in this work is the Keepaway domain (Stone et al., 2005), where several transfer learning methods have been applied recently (Taylor & Stone, 2005; Torrey et al., 2005; Hsu et al., 2004)

2. Policy Reuse

The goal of this section is to 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 bias in a new exploratory process. We also introduce a similarity concept between policies, which motivation is deeply described in (Fernández et al., 2005). Lastly, we describe the PRQ-learning algorithm (Fernández & Veloso, 2005).

2.1. Domain, Tasks and MDPs

A Markov Decision Process (Puterman, 1994) is represented with a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} >$, where $\mathcal{S}$ is the set of all possible states, $\mathcal{A}$ is the set of all possible actions, $\mathcal{T}$ is an unknown stochastic state transition function, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \Re$, and $\mathcal{R}$ is an unknown stochastic reward function, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \Re$. We focus on RL domains where different *tasks* can be solved. We introduce a task as a specific reward function, but the other concepts, $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{T}$ 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, $\mathcal{D}$, as a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{T} >$. We define a task, Ω, as a tuple $< \mathcal{D}, \mathcal{R}_\Omega >$, where $\mathcal{D}$ is a domain as defined before, and $\mathcal{R}_\Omega$ is the stochastic and unknown reward function.

In this work we assume that we are solving a task with absorbing goal states. Thus, if s_i is a goal state, $\mathcal{T}(s_i, a, s_i) = 1$, $\mathcal{T}(s_i, a, s_j) = 0$ for $s_i \neq s_j$, and $\mathcal{R}(s_i, a) = 0$, for all $a \in \mathcal{A}$. A trial starts by locating the learning agent in a random position in the environment. Each trial finishes when a goal state is reached or when a maximum number of steps, say H, is achieved. Thus, the goal is to maximize the expected average reinforcement per trial, say W, defined as $W = \frac{1}{K}\sum_{k=0}^{K}\sum_{h=0}^{H}\gamma^h r_{k,h}$, where γ $(0 \leq \gamma \leq 1)$ reduces the importance of future rewards, and $r_{k,h}$ defines the immediate reward obtained in the step h of the trial k, in a total of K trials. An action policy, $\Pi : \mathcal{S} \rightarrow \mathcal{A}$, defines for each state, the action to execute. The action policy Π^* is optimal if it maximizes the gain W in such a task, say W^*_Ω.

The goal of Policy Reuse is to describe how learning can be sped up if different policies, which solve different tasks, are used to bias the exploration process of the learning of the action policy of another similar task. Then, the scope of this work is summarized as: (i) we need to solve the task Ω, i.e. learn Π^*_Ω; (ii) we have previously solved the set of tasks $\{\Omega_1, \ldots, \Omega_n\}$, so we have the set of policies, $\{\Pi^*_1, \ldots, \Pi^*_n\}$, to solve them respectively; (iii) how can we use the previous policies, Π^*_i, to learn the new one, Π^*_Ω?

An efficient solution to this problem is the PRQ-Learning algorithm. This algorithm automatically answers two questions: (i) what policy, from the set $\{\Pi^*_1, \ldots, \Pi^*_n\}$, is used to bias the new learning process? (ii) once a policy Π_i is selected, how is it integrated in the learning process? The algorithm is based on an exploration strategy, π-reuse, which is able to bias the learning of a new policy with only one past policy. From this strategy, a similarity metric between policies is obtained, providing a method to select the most accurate policy to reuse. Both the π-reuse strategy and the similarity metric, defined in (Fernández et al., 2005), are summarized in the next subsection.

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, which is being learned currently. The π-reuse strategy follows the past policy, say Π_{past}, with a probability of ψ. However, with a probability of $1-\psi$, it exploits the new policy. Obviously, random exploration is always required, so when exploiting the new policy, it follows an ϵ-greedy strategy, as defined in Table 1. Lastly, the υ parameter allows the decay of the value of ψ in each trial.

Interestingly, the π-reuse strategy also contributes a similarity metric between policies, based on the gain obtained when reusing each policy. Let's call W_i the gain obtained while executing the π-reuse exploration strategy, reusing the past policy Π_i. We call Π^*_Ω the optimal action policy for solving the task Ω. W^*_Ω is the gain obtained when using the optimal policy, Π^*_Ω, to solve Ω. Therefore, W^*_Ω is the maximum gain that

π-reuse $(\Pi_{old}, K, H, \psi, \upsilon)$.
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, $a = \Pi_{old}(s)$
    With a probability of $1 - \psi_h$, $a = \epsilon$-greedy$(\Pi_{new}(s))$
    Receiv current state s', and reward, $r_{k,h}$
    Update $Q^{\Pi_{new}}(s, a)$, and therefore, Π_{new}
    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 and Π_{new}

Table 1. π-reuse Exploration Strategy.

can be obtained in Ω. Then, we can use the difference between W_Ω^* and W_i to measure the similarity among both policies using the distance metric shown in equation 1.

$$d_{\rightarrow}(\Pi_i, \Pi) = W_\Omega^* - W_i \tag{1}$$

In this case the distance metric is not symmetric, so $d_{\rightarrow}(\Pi_i, \Pi_j)$ could be different from $d_{\rightarrow}(\Pi_j, \Pi_i)$. This distance metric is also useful to estimate how useful to reuse the policy Π_i is to learn to solve the new task. Then, the most useful policy to reuse, from a set $\{\Pi_1, \ldots, \Pi_n\}$, is $\arg_{\Pi_i}\max(W_i), i = 1, \ldots, n$. Notice that W_Ω^* has disappeared of the formula, given that is independent of i. Thus, W_i, or the average reward obtained when reusing the policy Π_i with the π-reuse exploration strategy, 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. This idea is formalized in the PRQ-Learning algorithm.

2.3. PRQ-Learning Algorithm

The PRQ-Learning algorithm (Policy Reuse in Q-Learning) (Fernández & Veloso, 2005) is shown in Table 2. The learning algorithm used is Q-Learning (Watkins, 1989). It has been chosen because it is an off-policy algorithm, and therefore, it allows to learn a policy while following a different one. The goal is to solve a task Ω, i.e. to learn an action policy Π_Ω. We have n past policies to solve n different tasks respectively. For simplicity of the notation, we will call these policies $\Pi_1, \ldots, \Pi_n$. Let's call W_i the expected average reward that is received when reusing the policy Π_i with the π-reuse exploration strategy. Also, let's call W_Ω 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.

Policy Reuse in Q-Learning

- Given:
  1. A set of n policies, $\{\Pi_1^*, \ldots, \Pi_n^*\}$ to solve different tasks
  2. A new task Ω we want to solve
  3. A maximum number of trials to execute, K
  4. A maximum number of steps per trial, H
- Initialize:
  1. $Q_\Omega(s, a) = 0, \forall s \in \mathcal{S}, a \in \mathcal{A}$
  2. Initialize W_Ω to 0
  3. Initialize W_i to 0
  4. Initialize the number of trials where policy Π_Ω has been chosen, $U_\Omega = 0$
  5. Initialize the number of trials where policy Π_i has been chosen, $U_i = 0, \forall i = 1, \ldots, n$
- For $k = 1$ to K do
  - Choose an action policy, Π_j, randomly, assigning to each policy the probability of being selected computed by the following equation:
  $$P(\Pi_j) = \frac{e^{\tau W_j}}{\sum_{p=0}^{n} e^{\tau W_p}}$$
  - Initialize the state s to a random state
  - Set $R = 0$
  - for $h = 1$ to H do
    * Use Π_j to compute the next action to execute, a, following an exploitation strategy.
    * Execute a
    * Receive current state, s'
    * Receive current reward, r
    * Update $Q_\Omega(s, a)$ using Q-Learning update function:
    $$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a')]$$
    * Set $R = R + \gamma^h r$
    * Set $s \leftarrow s'$
  - Set $W_j = \frac{W_j U_j + R}{U_j + 1}$
  - Set $U_j = U_j + 1$
  - Set $\tau = \tau + \Delta\tau$

Table 2. PRQ-Learning

This algorithm has demonstrated to successfully reuse a predefined set of policies (Fernández & Veloso, 2005). In addition, algorithms to build the Policy Library has been proposed (**?**). However, as introduced at the begining, Policy Reuse requires that the action and state spaces of the different policies, and therefore, the transition function, are homogeneous. Next section describes how to apply Policy Reuse among policies learned in different state/action spaces by creating a mapping among them.

3. Policy Reuse Across Different State and Action Spaces

We assume that we have a past policy, Π_{past}. Π_{past} is a policy that solves the task $\Omega_{past} =< \mathcal{D}_{past}, \mathcal{R}_{past} >$, where $\mathcal{D}_{past} =< \mathcal{S}_{past}, \mathcal{A}_{past}, \mathcal{T}_{past} >$.

Now, we want to learn a new policy Π_{new} to solve a new task $\Omega_{new} =< \mathcal{D}_{new}, \mathcal{R}_{new} >$, where $\mathcal{D}_{new} =< \mathcal{S}_{new}, \mathcal{A}_{new} >$. We want to reuse the past policy, Π_{past} to learn the new problem Π_{new}. Given the policies are functions from states and actions, we only need to map these two elements, and we do not need to make any mapping between tasks, rewards and transition functions. Thus, we need to find a mapping ρ, that, given the policy Π_{past}, outputs a new policy $\hat{\Pi}_{past}$ that is executable in the domain $\mathcal{D}_{new}$

If the state and action spaces of Ω_{new} and Ω_{past} are homogeneous, then the past policies can be reused to bias the learning of the new task directly. In that case, $\hat{\Pi}_{past}(s) = \Pi_{past}(s)$. Then, the mapped policy $\hat{\Pi}_{past}(s)$ is added to the Policy Library and reused.

If the state and or action spaces are heterogeneous, we can not use the past policy directly in the new task. In that case, we need to find a mapping from the state/action spaces of the past task to the ones of the new one. That mapping is defined in equation 2. In that equation, $\rho_{\mathcal{A}} : \mathcal{A}_{past} \to \mathcal{A}_{new}$ is a function that maps the actions in the space $\mathcal{A}_{past}$ to actions in the space $\mathcal{A}_{new}$; and $\rho_{\mathcal{S}} : \mathcal{S}_{past} \to \mathcal{S}_{new}$ maps states in the space $\mathcal{S}_{past}$ to the space $\mathcal{S}_{new}$.

$$\hat{\Pi}_{past}(s) = \rho_{\mathcal{A}}(\Pi_{past}(\rho_{\mathcal{S}}(s))) \qquad (2)$$

The way to make this mapping, i.e. to define $\rho_{\mathcal{S}}$ and $\rho_{\mathcal{A}}$, depends on the source and the target tasks. For instance, this mapping can be done by an expert, which decides the correspondence among states and actions (Taylor & Stone, 2005; Torrey et al., 2005). That mapping can also be learned by observing to a mentor (Price & Boutilier, 2001). Last, the mapping may depend on the generalization method used to approximate the value function (Taylor & Stone, 2005). Next section describes the application of Policy Reuse on the Keepaway task, and how we make the mapping between different state spaces based on the function approximator used.

4. The Keepaway Domain

The Keepaway domain is a subdomain of robot soccer (Stone et al., 2005). It consists on two teams, the keepers and the takers. The behavior of the takers is fixed and defined a priori and consists on 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 on to learn a high level control of those skills for each of the keepers. The goal is to maximize the time that the keepers maintain the possesion of the ball in the lifelong term. The end condition for each episode is that the keepers looses the ball or that the ball goes out of a fixed sub-area of the field. Both conditions are also predefined.

There are several learning possibilities in this framework. Changing the number of keepers or takers, the size of the play area, the sensing capabilities of the agents, or the communication skills among them generate different learning problems. In the next section we describe the keepaway task as a Reinforcement Learning task, describing some details as the state and action spaces or the reward function. We follow the framework defined in (Taylor & Stone, 2005). Thus, each keeper learns its own behavior, so we assume that each of them is implemented as a reinforcement learning agent, as explained next.

4.1. Keepaway as a Reinforcement Learning Task

The state features and action possibilities use information derived from distances and angles between the keepers, the takers, and keepers and the center of the play area. Depending on the number of keepers and takers, the feautures used are different. Table **??** describes the compleate state space when 3 keepers play against 2 keepers (3 vs. 2), and when 4 keepers play against 3 keepers (4 vs. 3). In this case, 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 in each moment.

Thus, the number of features for the different versions of the Keepaway are 13, 19 and 25 respectivelly. 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. Thus, the complete set of actions for the different number of keepers and takers is defined in Table 4.1.

As the table shows, the number of actions also grows with the number of keepers. In this version of the Keepaway task, the only decission is taken when the player holds the ball. If that cases, it needs to decide among the previously defined actions. If the player

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

Table 3. State spaces of the different keepaway tasks.

3 vs. 2	4 vs. 3	5 vs. 4
$Hold$	$Hold$	$Hold$
$Pass(k_2)$	$Pass(k_2)$	$Pass(k_2)$
$Pass(k_3)$	$Pass(k_3)$	$Pass(k_3)$
	$Pass(k_4)$	$Pass(k_4)$
		$Pass(k_5)$

Table 4. Actions in the different keepaway tasks.

does not hold the ball, it executes a predefined behavior as well as the takers do.

The reward is the time that the team holds the ball since the agent executed the last action. Thus, the reward does not depend only of the executed action, but also of the actions executed by the other agents, what is called *inherently cooperative rewards*. As in other different tasks as Cooperative... (CMOMMT) (), the cooperative behaviors is expected to emerge from the individual behavior learned by each agent independently of the others.

The action space in the Keepaway task is reduced to a few actions. However, state space is composed of several features that can be considered continuous. Thus, a method for state space generalization is required in order to improve learning capabilities. Some methods for function approximation, as CMAC or neural networks, have been applied for the keepaway (Taylor et al., 2005). The next section describes the application of the VQQL model for discretizing the keepaway state space.

4.2. Function Approximation in the Keepaway Task

The VQQL algorithm (Vector Quantization for Q-Learning) (Fernández & Borrajo, 2000; Fernández & Parker, 2001), is based on the unsupervised discretization of the state space. It applies vector quantization methods, and more specifically, the Generalized Lloyd Algorithm (GLA) (Lloyd, 1982), also called k-means. This method generates a set of prototypes, that together with the nearest neighbor rule, defines Voronoi regions (Gersho & Gray, 1992). Each of these regions clusters a set of states of the original state space representation. Figure 1 shows how to represent the action policy following this approach.

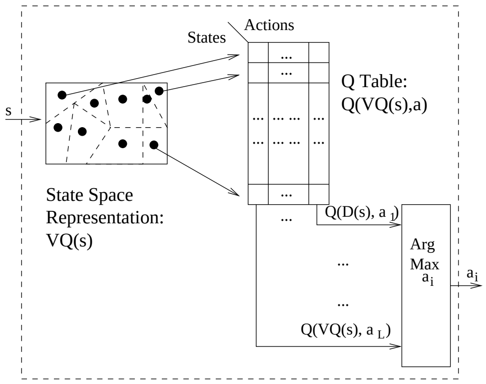

Figure 1. Function approximation with VQQL.

The use of vector quantization and the Generalized Lloyd Algorithm to solve the generalization problem in RL algorithms requires two consecutive phases:

Learning the quantizer. Designing the vector quantizer, $VQ(s)$, composed of B prototypes from the input data obtained from the environment during exploration.

Learning the Q function. Once the vector quantizer is designed (i.e., the environment is clustered into B different states), the Q function must be learned, generating the Q table composed of B rows and a column for each action.

On-line VQQL

- Obtain a set P of examples of states.
- Design a vector quantizer (discretization function) $VQ(s)$ using P with the Generalized Lloyd Algorithm
- Learn the Q function using the following steps:
  - Choose an action following an exploration/exploitation strategy. Receive the reward, obtaining an experience tuple $< s, a, s', r >$
  - Quantify the experience tuple, obtaining $< \hat{s}, a, \hat{s'}, r >$, where $\hat{s} = VQ(s)$ and $\hat{s'} = VQ(s')$
  - Update the Q table following the Q-Learning update equation

Figure 2. On-line VQQL algorithm.

The Vector Quantization technique takes advantage of the statistical characteristics of the domain to reduce its size, considering only relevant areas from the whole domain. The algorithm has been applied to learn the ball interception skill of a goalie (Fernández & Borrajo, 2000), and in Cooperative Multi-robot Observation of Multiple Moving Targets (CMOMMT) (Parker, 2002), in order to obtain collaborative behaviors in a multi-robot observation task (Fernández & Parker, 2001; Parker et al., 2002). Those experiments showed that the nearest prototype approach used to discretize the state space produces better results than uniform discretizations.

In the following sections, the VQQL model has been applied to solve the Keepaway. We will also show that this method allows to transfer policies very easily.

5. Learning in the Keepaway

In this section, we describe the experiments performed on the Keepaway domain. We differentiate three different tasks. Firstly the 3vs2-keepaway, where 3 keepers play against two players; secondly, the 4vs3-keepaway; and last, the 5vs4-keepaway, where 5 keepers play against 4 takers. The three tasks are learned in the order that have been defined. We define that there are two possibilities. 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, reuses the acquired policy to solve the 4vs3-keepaway. The agents 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 for keepers 1, 2, and 3; 1 for keeper 4, and none for keeper 5), as defined in Figure ??

	3vs2-keepaway	4vs3-keepaway	5vs4-keepaway
Keeper 1	Learn Π^{k1}_{3vs2} from scratch	Learn Π^{k1}_{4vs3} by reusing $L^{k1} = \{\Pi^{k1}_{3vs2}\}$	Learn Π^{k1}_{5vs4} by reusing $L^{k1} = \{\Pi^{k1}_{3vs2}, \Pi^{k1}_{4vs3}\}$
Keeper 2	Learn Π^{k2}_{3vs2} from scratch	Learn Π^{k2}_{4vs3} by reusing $L^{k2} = \{\Pi^{k2}_{3vs2}\}$	Learn Π^{k2}_{5vs4} by reusing $L^{k2} = \{\Pi^{k2}_{3vs2}, \Pi^{k2}_{4vs3}\}$
Keeper 3	Learn Π^{k3}_{3vs2} from scratch	Learn Π^{k3}_{4vs3} by reusing $L^{k3} = \{\Pi^{k3}_{3vs2}\}$	Learn Π^{k3}_{5vs4} by reusing $L^{k3} = \{\Pi^{k3}_{3vs2}, \Pi^{k3}_{4vs3}\}$
Keeper 4	Not Playing	Learn Π^{k4}_{4vs3} from scratch	Learn Π^{k4}_{5vs4} by reusing $L^{k4} = \{\Pi^{k4}_{4vs3}\}$
Keeper 5	Not Playing	Not Playing	Learn Π^{k5}_{5vs4} from scratch

Table 5. Description of the tasks solved. The first one is 3vs2-keepaway, the second one is 4vs3-keepaway reusing the policies acquired in the 3vs2-keepaway. The last one is 4vs-4 keepaway reusing the policies acquired in both 3vs2-keepaway and 4vs3-keepaway.

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.

5.1. Policy Reuse in the Keepaway

The 3vs2-keepaway is a task, $\Omega^{3vs2} =< \mathcal{D}^{3vs2}, \mathcal{R} >$, defined in the domain $\mathcal{D}^{3vs2}$, with a reward function $\mathcal{R}$. The domain is defined as a tuple, $\mathcal{D}^{3vs2} =< \hat{\mathcal{S}}^{3vs2}, \mathcal{A}^{3vs2}, \mathcal{T}^{3vs2} >$. $\hat{\mathcal{S}}^{3vs2}$ is a discretized version of the original state space $\mathcal{S}^{3vs2}$,. The discretization is performed by a function $VQ^{3vs2} : \mathcal{S}^{3vs2} \rightarrow \hat{\mathcal{S}}^{3vs2}$. $\mathcal{S}^{3vs2}$ and $\mathcal{A}^{3vs2}$ were defined in tables ?? and ?? respectively. Both, the transition function $\mathcal{T}^{3vs2}$ and the reward function are unknown for the agent. The goal is to learn an action policy $\Pi_{\Omega^{3vs2}} : \hat{\mathcal{S}}^{3vs2} \rightarrow \mathcal{A}^{3vs2}$ that outputs actions, given any state of the discretized state space.

The 4vs3-keepaway task is formalized as following:

- $\mathcal{D}^{4vs3} =< \hat{\mathcal{S}}^{4vs3}, \mathcal{A}^{4vs3}, \mathcal{T}^{4vs3} >$ where $\hat{\mathcal{S}}^{4vs3} = VQ^{4vs3}(\mathcal{S}^{4vs3})$
- $\Omega^{4vs3} =< \mathcal{D}^{4vs3}, \mathcal{R} >$
- $\Pi_{\Omega^{4vs3}} : \hat{\mathcal{S}}^{4vs3} \rightarrow \mathcal{A}^{4vs3}$

Then, the mapping from a policy in the 3vs2-keepaway to the 4vs3-keepaway is performed as defined in equation 3:

$$\Pi_{\Omega^{4vs3}}(s) = \Pi_{\Omega^{3vs2}}(VQ^{3vs2}(P^{4vs3}_{3vs2}(s))) \tag{3}$$

where $P^{4vs3}_{3vs2}(s)$ is a function $P^{4vs3}_{3vs2} : \mathcal{S}^{4vs3} \rightarrow \mathcal{S}^{3vs2}$ that, given a state in the 4vs3-keepaway state space, $\mathcal{S}^{4vs3}$, projects it on the 3vs2-keepaway state space, $\mathcal{S}^{3vs2}$.

In the keepaway task, this proyection is performed very easily, and is derived from the semantic of the features, as used in other different works (Taylor & Stone, 2005; Camacho et al., 2005). ???

5.2. Results

In the experiments, we have used the Keepaway layer Framework 0.6 [1]. As in other transfer learning paper that uses this simulator as testbed (Taylor & Stone, 2005), the field size is 25×25, vision capabilities are set to full, and the synchronous mode is set on to speed up the simulator. However, different simulator versions have different low level implementations of the skills, so direct comparisons with previously published results can not be performed, given that the simulation version used on (Taylor & Stone, 2005) is not longer available. However, we will try to provide qualitative comparisons as far as it is possible. [2] In order to have some baseline results, in all the cases we introduce the performance of two policies, the random action selection strategy, and a policy where the agents always passes to the second keeper (which outperforms the policy where the agents always hold the ball, used for comparisons in other papers (Stone et al., 2005)).

In the next subsections we provide the results obtained when solving the Keepaway tasks, following the steps described in Table 5. Thus, we first learn the 3vs2-keepaway. Then, we solve the 4vs3-keepaway by transfering the policies learned in the 3vs2-keepaway. Last, we solve the 5vs4-keepaway by reusing the policies learned in the 3vs2 and 4vs3-keepaway. In all the cases, the size of the discretized state space is 512 states. The state space discretization is learned with the GLA algorithm over data generated after 2000 episodes, where all the agents followed a random action selection strategy. All the previous values have been chosen as a result of an informal experimentation. Thus, they may not be optimal for the task that we are solving, but provide us with results that are accurate enought to make or study on Policy Reuse, without evaluating in depth the influence of the discretizations obtained in the performance of the keepaway task.

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

5.2.1. 3vs2-keepaway

Firstly, we have learned the 3vs2-keepaway. The results are summarized in Table fig:3vs2-results. The x axis of the figure describes the training time, while the 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 11 seconds.

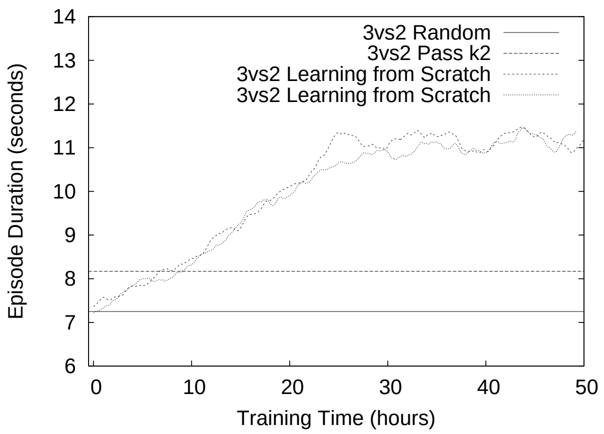

Figure 3. Results of learning in the 3vs2-keepaway

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

[2] Thanks to Peter Stone, Matthew Taylor, Yaxin Liu and Gregory Kuhlmann for their valuable comments about their works on the Keepaway task.

5.2.2. 4VS3-KEEPAWAY

After learning the 3vs2-keepaway, the agents received the task 4vs3-keepaway. In this case, we followed two different approaches for learning: learning from scratch, and Policy Reuse, following the scheme introduced in Table ??. The results are summarized in Figure 5. 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 much harder.

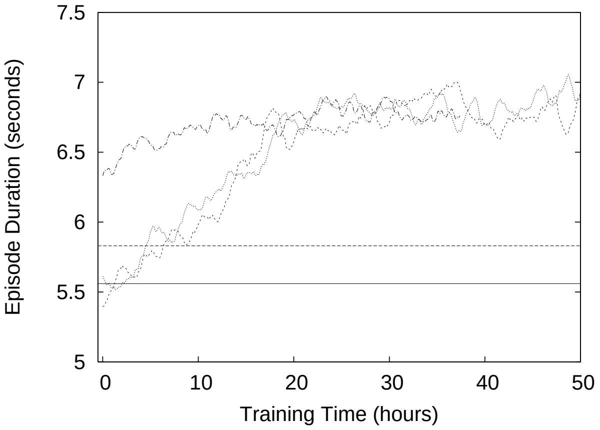

Figure 4. Results of learning in the 4vs3-keepaway

When learning, we can differentiate in 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 behaviour (around 5.5) up to more than 6.5. However, when the agents reuses the policy learned in the 3vs2-keepaway through Policy Reuse, the initial value is around 6.4, growing up to more than 6.5 in around 11 ours. This demonstrates that reusing the past policy improves the behaviour of the learning agent since the early steps of the learning. The second phase, after 20 hours of learning, does not show significant differences in the learning curves.

This result is qualitatively different to the one reported on value function based behavior transfer (Taylor & Stone, 2005). In that work, both learning from scratch and behaviour 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 4vs-3 keepaway, they need to initialize some values of the Q-function by hand. That means that, for instance, the Q values of the action "Pass k3" in the 3vs2-keepaway is transferred both to the action "Pass k3" and to the action "Pass k4" of 4vs3-keepaway. This initialization of the Q values are required to make the Q table transferred more homogeneous than if some values are set to 0. Furthermore, the results obtained may vary depending on how that initialization is performed. 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 (Taylor & Stone, 2005) are different, since the authors report that they obtain values of 9 for the performance, both when learning from scratch and when transfering behaviors. However, given that the differences appear also when learning from scratch, they must be due to other variables besides the transfer lerarning, as the simulator version, the learning algorithms (Sarsa instead of Q-Learning) or the generalization methods used (CMAC instead of VQQL).

5.2.3. 5VS4-KEEPAWAY

After learning the 3vs2 and 4vs3-keepaway, the situation of the agents, as defined in Table ?? is different. 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 forth agent can reuse only the policy of the 4vs3-keepaway, given that it did not participated in the 3vs2-keepaway. The fifth keeper never played before, so it needs to learn from scratch. The results are shown in Figure 5.

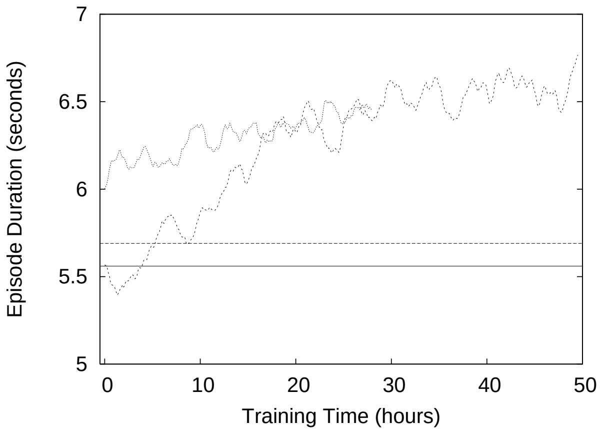

Figure 5. Results of learning in the 5vs4-keepaway

References

Camacho, D., Fernndez, F., & Rodelgo, M. A. (2005). Roboskeleton: An architecture for coordinating robot soccer agents. *Engineering Applications of Artificial Intelligence*.

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

Fernández, F., & Borrajo, D. (2000). VQQL. Applying vector quantization to reinforcement learning. In *Robocup-99: Robot soccer world cup iii*, no. 1856 in Lecture Notes in Artificial Intelligence, 292–303. Springer Verlag.

Fernández, F., Borrajo, D., & Parker, L. (2005). A reinforcement learning algorithm in cooperative multi-robot domains. *Journal of Intelligent Robotics Systems*.

Fernández, F., & Parker, L. (2001). Learning in large cooperative multi-robot domains. *International Journal of Robotics and Automation*, *16*, 217–226.

Fernández, F., & Veloso, M. (2005). *Exploration and policy reuse* (Technical Report CMU-CS-05-172). School of Computer Science, Carnegie Mellon University.

Gersho, A., & Gray, R. M. (1992). *Vector quantization and signal compression*. Kluwer Academic Publishers.

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

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

Lloyd, S. P. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory* (pp. 127–135).

Parker, L. E. (2002). Distributed algorithms for multi-robot observation of multiple moving targets. *Autonomous Robots*, *12*, 231–255.

Parker, L. E., Touzet, C., & Fernández, F. (2002). *Robot teams: from diversity to polymorphism*, chapter Techniques for Learning in Multi-Robot Teams, 191–236. A. K. Peters Publishers Ltd.

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

Puterman, M. L. (1994). *Markov decision processes - discrete stochastic dynamic programming*. New York, NY.: John Wiley & Sons, Inc.

Stone, P., Sutton, R. S., & Kuhlmann, G. (2005). Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*. To appear.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. Cambridge, Massachusetts: MIT Press.

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

Taylor, M. E., & Stone, P. (2005). Behavior transfer for value-function-based reinforcement learning. *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*. To appear.

Taylor, M. E., Stone, P., & Liu, Y. (2005). Value functions for RL-based behavior transfer: A comparative study. *Proceedings of the Twentieth National Conference on Artificial Intelligence*. To appear.

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

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

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. Doctoral dissertation, King's College, Cambridge, UK.