Abstract

Transfer learning refers to the process of conveying experience from a simple task to another more complex (and related) task in order to reduce the amount of time that is required to learn the latter task. Typically, in a transfer learning procedure the agent learns a behavior in a source task, and it uses the gained knowledge in order to speed up the learning process in a target task. Reinforcement Learning algorithms are time expensive when they learn from scratch, especially in complex domains, and transfer learning comprises a suitable solution to speed up the training process. In this work we propose a method that decomposes the target task in several instances of the source task and uses them to extract an advised action for the target task. We evaluate the efficacy of the proposed approach in the robotic soccer Keepaway domain. The results demonstrate that the proposed method helps to reduce the training time of the target task.

1 Introduction

Transfer learning refers to the process of conveying experience from a simple task to another more complex (and related) task in order to reduce the amount of time that is required to learn the latter task. The procedure of transfer learning encounters very often in the human behavior and it has been studied comprehensively in psychology [14, 4]. For example, if we first learn how to ride a bike it will be easier then to learn how to ride a motorbike.

Typically, in a transfer learning procedure the agent learns a behavior in a source task, and then it uses the gained knowledge in order to speed up the learning process in a target task. These tasks can belong either in the
same domain, for example two mazes that differ in their size or structure, or in
different domains like grid world and robot soccer. The more similar the two
tasks are, the easier it is to transfer experience between them.

Recently, transfer learning among Reinforcement Learning (RL) agents has
received a lot of attention. This is due to the fact that RL algorithms are time
expensive when they learn from scratch, especially in complex domains, and
transfer learning comprises a suitable solution to speed up the training process.

Several approaches have been proposed in the past for transfer learning. In [6]
a set of elemental task are composed in order to solve quicker a larger
task while in [1] the proposed method automatically detects subtasks that are
represented as graphs, which are used to initialize the learned function in the
target task. In [2], an algorithm reuses policies from previous learned source
tasks. Methods that make use of rules exported from the experience gained in
a source task have been proposed in [3, 15, 12].

In this work we propose a novel method for transfer learning among RL
agents, which is based on the decomposition of the target task in several different
instantiations of the source task. Note, that the way that we decompose the
target task differs with previous approaches as we find all the valid instances of
the source task that exist in the target task. The identified source instances are
used in order to provide an advice to the target task.

We evaluate the proposed approach in the Keepaway domain [9]. Keepaway
is a subset of the Robocup Soccer domain, where a number of keepers try to
hold the ball as long as possible, while a team of takers try to take possession
of the ball. The empirical evidence show that the proposed approach reduces
the complete training time in the target task. Additionally, experiments we
conducted in large task show that the proposed approach scales well with respect
to the size of the problem.

The rest of the paper is structured as follows. Section 2 presents background
information in Reinforcement Learning. Section 3 describes the domain that will
be used for investigating the efficacy of the proposed approach that is introduced
in Section 4. In Section 5 we describe the experimental setup and present the
results. Section 6 reviews related work on transfer learning and finally Section
7 concludes this work.

2 Background

Reinforcement learning (RL) is a machine learning technique that was grounded
from the theory of psychology. It concerns with the task of how an autonomous
agent that interacts with a dynamic environment can learn an optimal behavior
in order to maximize a long-term reward that is accorded by the environment.

The commonly used model for the RL problems is the Markov Decision
Process (MDP) [5] which is defined as a tuple \((S, A, R, T)\) where \(S\) is the finite
set of possible states, \(A\) is the finite set of possible actions, \(R : S \times A \rightarrow \mathbb{R}\) is a
reward function that returns a real value \(r\) that is received by the agent as an
outcome of taking an action \(a \in A\) in a state \(s \in S\). Finally, \(T : S \times A \times S \rightarrow [0, 1]\)
is the state transition probability function which denotes the probability of moving to a state \( s' \) after executing action \( a \) in state \( s \).

The objective of the agent is to find a policy \( \pi : S \rightarrow A \), which denotes how the agent acts in a certain situation, that maximizes the reward received over time. This policy is called optimal and is denoted by \( \pi^* \). A way to find the optimal policy is to first learn the \( Q \) value-function which is defined for each pair of \( s, a \) and denotes the expected accumulated reward over time for executing action \( a \) in state \( s \).

A well-known algorithm for learning \( Q \) is Sarsa(\( \lambda \)) [10] which approximates the \( Q \)-function with the following form:

\[
Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + Q(s', a'))
\]

where \( r \) is the immediate reward received after executing action \( a \) in state \( s \) and \( s', a' \) are the successor state and the chosen action respectively. Finally, \( \alpha \) is the learning rate.

3 The Keepaway Testbed

In this section we shortly describe the Keepaway domain and standard task that is used for evaluating the proposed method and for giving some concrete examples. Further details about the domain can be found in [9].

Keepaway is a subset of the RoboCup robot soccer domain, where \( K \) keepers try to hold the ball for as long as possible, while \( T \) takers (usually \( T = K - 1 \)) try to intercept the ball. The agents are placed within a fixed region at the start of each episode, which ends when the ball leaves this region or the takers intercept it.

For the purpose of defining a machine learning testbed, a standard task was introduced, concerning the decision-making of keepers when they possess the ball. The task is modeled as a semi-Markov decision process (SMDP), as it defines macro-actions that may last for several time steps. The available macro-actions for a keeper with ball possession are \( \text{HoldBall} \) and \( \text{Pass-k-ThenReceive} \), where \( k \) is another keeper. A keeper executing \( \text{HoldBall} \) remains stationary. A keeper executing \( \text{Pass-k-ThenReceive} \) performs a series of actions in order to pass the ball to teammate \( k \) and then executes the \( \text{Receive} \) sub-policy: If no keeper possesses the ball and he is the fastest keeper to the ball then he executes macro-action \( \text{GoToBall} \), otherwise he executes macro-action \( \text{GetOpen} \) in order to move to an open area. \( \text{Receive} \) is also executed by keepers when they don’t possess the ball.

The features that describe the state of the environment for a keeper \( K_1 \) that possesses the ball are: a) the distances of all agents to the center of the field, b) the distance of all other players to \( K_1 \), c) the distances of \( K_1 \)'s teammates to the closest opponent, and d) the minimal angles between \( K_1 \)'s teammates and an opponent with the vertex at \( K_1 \). Figure 1 depicts the 13 state variables for the 3vs2 game, where keepers are represented as circles and takers as triangles.
Figure 1: The 13 state variables of the 3vs2 Keepaway game.

Note that keepers and takers are indexed with numbers from 1 to $K$ and 1 to $T$ respectively, according to ball distance.

The task becomes harder as extra keepers and takers are added to the fixed-sized field, due to the increased number of state variables on one hand, and the increased probability of ball interception in an increasingly crowded region on the other.

4 The Proposed Approach

One of the key novel contributions, and at the same time a basic assumption, of the proposed approach is the observation that in certain domains the target task (states and actions) can be mapped to a number of different instantiations (states and actions) of the source task.

In Keepaway, any $K^t$ vs $T^t$ task can be mapped to a number of different instantiations of a $K^s$ vs $T^s$ task, where $K^s < K^t$ and $T^s < T^t$, simply by deleting $K^t - K^s$ teammates and $T^t - T^s$ opponents of the keeper with ball possession. The actual number of instantiations is:

$$\frac{(K^t - 1)!T^t!}{(K^s - 1)![(K^t - K^s)!T^s!(T^t - T^s)!]}$$

For example, a pair $(s_t, A^{s_t})$ of state and actions in the 4vs3 target task can be mapped to 9 different $(s_s, A^{s_s})$ pairs of state and actions in the 3vs2 source task. Figure 2 shows two examples of such mappings. In the first instance $K_2$ and $K_3$ are $K_1$’s teammates and $T_1$ and $T_2$ are its opponents. In the second instance the teammates are $K2$ and $K_4$ and the opponents are $T_1$ and $T_3$. Note that the mapping of state and actions is implicit, as both states and actions are defined according to agent distances and angles.
Figure 2: The 4vs3 task (a) is mapped to 2 instantiations of the 3vs2 task (b,c). Gray players are neglected in the 3vs2 task instantiations.

We believe that such mappings can be established in a similar fashion between tasks of other multi-agent domains, which differ only in the number of agents. We also give an example of such a mapping between the 3-dimensional (3D) and the 2-dimensional (2D) tasks in the *mountain car* single-agent domain [7]. In the 2D task, the state is composed of the horizontal position $x$ and velocity $v_x$ and the actions are \{Neutral, Left, Right\}. In the 3D task, the set of state variables is expanded with the position $y$ and velocity $v_y$ for the new dimension, while the new action set becomes \{Neutral, West, East, South, North\}. A mapping can be established between the two tasks, by ignoring either the $x$ or $y$ dimension.

The idea of mapping the target task to all possible instantiations of the source task could be utilized in different ways for transfer learning. For example,
one could copy the weights from the best instance (according to some criterion) to the target task. We hypothesize that the different mappings will increase the effectiveness of transfer learning algorithms, compared to a single mapping [13, 12, 15].

In this work we propose an approach that receives an action advice based on the source instances. The basic steps of the proposed approach can be summarized as follows:

- Decompose the target task to valid instances of the source task.
- Extract an advice from the instances
- According to a strategy follow or not the advice

### 4.1 Mapping the target task

Formally, given that the target task can be mapped to $N$ instances of the source task, the specification of $N$ pairs of mapping functions is required. Each pair $(f^1_S, f^1_A), i = 1\ldots N$ consists of a function $f^i_S : S_{target} \rightarrow S_{source}$ mapping the set of target states, $S_{target}$, to the set of source states, $S_{source}$, and a function $f^i_A : A_{target} \rightarrow A_{source}$ that partially maps the set of target actions $A_{target}$ to the set of source actions $A_{source}$. By partially, we mean that certain actions in the target domain, might not have a corresponding action in the source domain in some of the instantiations. Figure 3 sketches the mapping of a target task to $N$ different instantiations of the source task through the corresponding functions.

![Figure 3: Schematic mapping of the target task. For each instance of the source task, a pair of mapping functions is defined.](image)

Mapping the target task to several instances of the source task is a domain-dependent process and requires the involvement of a user. In the rest of this subsection we define two pairs of mapping functions, one for Keepaway and one for mountain-car.

In Keepaway, we examine the mapping of the 4vs3 task to the 3vs2 task, according to the instantiation in Figure 2(c). Table 1 shows a partial mapping between the state variables of the two tasks, which can be used to define the
complete mapping function \( f_S \) between the states of the two tasks. Table 2 shows the partial mapping function \( f_A \) between the actions of the two tasks. Note that both functions can be automatically defined for all instantiations, based on the subset of agents that are present in the corresponding instantiation.

<table>
<thead>
<tr>
<th>4 vs. 3 variable</th>
<th>3 vs. 2 variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{dist}(K_1, C) )</td>
<td>( \text{dist}(K_1, C) )</td>
</tr>
<tr>
<td>( \text{dist}(K_1, K_2) )</td>
<td>( \text{dist}(K_1, K_2) )</td>
</tr>
<tr>
<td>( \text{dist}(K_1, K_3) )</td>
<td>( \text{dist}(K_1, K_3) )</td>
</tr>
<tr>
<td>( \text{dist}(K_1, K_4) )</td>
<td>( \text{dist}(K_1, K_3) )</td>
</tr>
<tr>
<td>( \text{dist}(K_1, T_1) )</td>
<td>( \text{dist}(K_1, T_1) )</td>
</tr>
<tr>
<td>( \text{dist}(K_1, T_2) )</td>
<td>( \text{dist}(K_1, T_2) )</td>
</tr>
<tr>
<td>( \text{dist}(K_2, C) )</td>
<td>( \text{dist}(K_2, C) )</td>
</tr>
<tr>
<td>( \text{dist}(K_3, C) )</td>
<td>( \text{dist}(K_3, C) )</td>
</tr>
<tr>
<td>( \text{min}(\text{dist}(K_2, T_1), \text{dist}(K_2, T_2)) )</td>
<td>( \text{min}(\text{dist}(K_2, T_1), \text{dist}(K_2, T_2)) )</td>
</tr>
<tr>
<td>( \text{min}(\text{dist}(K_3, T_1), \text{dist}(K_3, T_2)) )</td>
<td>( \text{min}(\text{dist}(K_3, T_1), \text{dist}(K_3, T_2)) )</td>
</tr>
<tr>
<td>( \text{min}(\text{dist}(K_4, T_1), \text{dist}(K_4, T_2)) )</td>
<td>( \text{min}(\text{dist}(K_4, T_1), \text{dist}(K_4, T_2)) )</td>
</tr>
<tr>
<td>( \text{min}(\text{dist}(K_2, T_1), \text{dist}(K_2, T_2)) )</td>
<td>( \text{min}(\text{dist}(K_2, T_1), \text{dist}(K_2, T_2)) )</td>
</tr>
<tr>
<td>( \text{min}(\text{dist}(K_3, T_1), \text{dist}(K_3, T_2)) )</td>
<td>( \text{min}(\text{dist}(K_3, T_1), \text{dist}(K_3, T_2)) )</td>
</tr>
<tr>
<td>( \text{min}(\text{dist}(K_4, T_1), \text{dist}(K_4, T_2)) )</td>
<td>( \text{min}(\text{dist}(K_4, T_1), \text{dist}(K_4, T_2)) )</td>
</tr>
<tr>
<td>( \text{ang}(K_2, K_1, T_1) )</td>
<td>( \text{ang}(K_2, K_1, T_1) )</td>
</tr>
<tr>
<td>( \text{ang}(K_2, K_1, T_2) )</td>
<td>( \text{ang}(K_2, K_1, T_2) )</td>
</tr>
<tr>
<td>( \text{ang}(K_2, K_1, T_3) )</td>
<td>( \text{ang}(K_2, K_1, T_2) )</td>
</tr>
<tr>
<td>( \text{ang}(K_3, K_1, T_1) )</td>
<td>( \text{ang}(K_3, K_1, T_1) )</td>
</tr>
<tr>
<td>( \text{ang}(K_3, K_1, T_2) )</td>
<td>( \text{ang}(K_3, K_1, T_2) )</td>
</tr>
<tr>
<td>( \text{ang}(K_3, K_1, T_3) )</td>
<td>( \text{ang}(K_3, K_1, T_2) )</td>
</tr>
<tr>
<td>( \text{ang}(K_4, K_1, T_1) )</td>
<td>( \text{ang}(K_4, K_1, T_1) )</td>
</tr>
<tr>
<td>( \text{ang}(K_4, K_1, T_2) )</td>
<td>( \text{ang}(K_4, K_1, T_2) )</td>
</tr>
<tr>
<td>( \text{ang}(K_4, K_1, T_3) )</td>
<td>( \text{ang}(K_4, K_1, T_2) )</td>
</tr>
</tbody>
</table>

Table 1: A mapping of the state variables of the 4vs3 target task to the variables of the 3vs2 source task for the instantiation of Figure 2(c).

In mountain-car, we examine the mapping of the 3D task to the 2D task for the case where we ignore the \( x \) dimension. Table 4.1 shows the corresponding mapping of state variables and actions.

### 4.2 Extracting Advice

Having defined the functions that establish a mapping from a target task to several instantiations of the source task, the next step is to use the knowledge acquired in the source task to improve the learning procedure in the target task. In order to accomplish this, the proposed method uses the experience gained from the source task to extract an advice for the target task.

We assume that the RL agent has been trained in the source task and that it has access to a function \( Q'(s', a') \) returning an estimation of the \( Q \) value for a
<table>
<thead>
<tr>
<th>4 vs. 3 action</th>
<th>3 vs. 2 action</th>
</tr>
</thead>
<tbody>
<tr>
<td>HoldBall</td>
<td>HoldBall</td>
</tr>
<tr>
<td>Pass-K2-ThenReceive</td>
<td>Pass-K2-ThenReceive</td>
</tr>
<tr>
<td>Pass-K3-ThenReceive</td>
<td>-</td>
</tr>
<tr>
<td>Pass-K4-ThenReceive</td>
<td>Pass-K3-ThenReceive</td>
</tr>
</tbody>
</table>

Table 2: The function that partially maps the actions of the 4vs3 target task to the actions of the 3vs2 source task for the instantiation of Figure 2(c).

<table>
<thead>
<tr>
<th>3D variable</th>
<th>2D variable</th>
<th>3D action</th>
<th>2D action</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-</td>
<td>Neutral</td>
<td>Neutral</td>
</tr>
<tr>
<td>y</td>
<td>x</td>
<td>West</td>
<td>-</td>
</tr>
<tr>
<td>vx</td>
<td>-</td>
<td>East</td>
<td>-</td>
</tr>
<tr>
<td>vy</td>
<td>vy</td>
<td>South</td>
<td>Left</td>
</tr>
<tr>
<td></td>
<td></td>
<td>North</td>
<td>Right</td>
</tr>
</tbody>
</table>

Table 3: A pair of mapping functions for the mountain car domain.

state $s'$ and action $a'$ of the source task. The agent is currently being trained in the target task, learning a function $Q(s, a)$ that approximates the $Q$ function, and senses the state $s$. The action that will be executed by the agent is ruled by the $\epsilon - Advice$ procedure which is depicted in Algorithm 1. This is a variation of the well known $\epsilon - greedy$ rule which is used to balance the exploration and the exploitation in RL algorithms.

According to the $\epsilon - Advice$ strategy the learning agent can select to explore the state space, to exploit the current knowledge or to exploit the experience from the source task. In lines 9 to 19 the algorithm extracts the advised action from the source task.

More specifically, for each instance $i$ of the source task that is recognized in the target task, the corresponding mapping function $f^i_s$ is used to transform the target state $s$ to its source representation $s'$. Then for each available action in the state of the source task, the corresponding $Q$-values are computed. The computation of the $Q$-values depends on the function approximation method that is used.

As the algorithm iterates over the instances, it stores the maximum $Q$-value, $Q'_{\text{max}}$ and the corresponding action $a_{\text{max}}$, along with the index, $i_{\text{max}}$, of the instance that they correspond to.

After the advised action is extracted, it is transformed to its target representation using a mapping function $g_A$, which is an inverse function of $f_A$ and maps a source action to its equivalent target action. After that, the algorithm checks (lines 21-25) whether to act according to the current learned policy or to follow the action that is suggested by the source task.

The agent must prefer the advised action if the difference between the $Q'_{\text{max}}$ value and the maximum value of the current learned $Q$-function, $Q(s, a_{\text{cur}})$ is greater than the value of the current learned $Q$-function for the advised action.
Algorithm 1 The $\epsilon$-Advice strategy.

1: procedure $\epsilon$-ADVICE($\epsilon, s, f_s, A, Q, Q'$)
2:   \hspace{1em} $p \leftarrow \text{RandomReal}(0, 1)$
3:   \hspace{1em} if $p \leq \epsilon$ then
4:       \hspace{2em} return random action
5:   \hspace{1em} else
6:     \hspace{2em} $Q'_{\text{max}} \leftarrow 0$
7:     \hspace{2em} $a'_{\text{max}} \leftarrow \emptyset$
8:     \hspace{2em} $i_{\text{max}} \leftarrow 0$
9:     \hspace{1em} for $i \leftarrow 1 \ldots N$ do
10:        \hspace{2em} $s' \leftarrow f_s^i(s)$
11:          \hspace{2em} for all $a' \in A'_{\text{source}}$ do
12:             \hspace{3em} if $Q'_{\text{max}} < Q'(s', a')$ then
13:                \hspace{4em} $Q'_{\text{max}} \leftarrow Q'(s', a')$
14:                \hspace{4em} $i_{\text{max}} \leftarrow i$
15:                \hspace{4em} $a'_{\text{max}} \leftarrow a'$
16:          \hspace{2em} end if
17:     \hspace{1em} end for
18:   \hspace{1em} end for
19:     \hspace{1em} $a_{\text{adv}} \leftarrow g_A^{i_{\text{max}}}(a'_{\text{max}})$
20:     \hspace{1em} $a_{\text{cur}} \leftarrow \text{arg max}_a Q(s, a)$
21:     \hspace{1em} if $Q'_{\text{max}} - Q(s, a_{\text{cur}}) > Q(s, a_{\text{adv}})$ then
22:       \hspace{2em} return $a_{\text{adv}}$
23:     \hspace{1em} else
24:       \hspace{2em} return $a_{\text{cur}}$
25:   \hspace{1em} end if
26: end procedure

$Q(s, a_{\text{adv}})$.

In the initial stages of learning, the agent will be biased to prefer the recommended actions depending on the impact of the source task $Q$-values. The impact of the source task depends strongly on how much time we spent in training it. More specifically, if the source task was trained for a small number of episodes then the $Q$-values will be also small and afterwards the agent in the target task will use the advised actions for a small period. As learning proceeds, the values of the target $Q$-function will increase and the initial bias will be overridden.

Training the source task in more episodes will cause greater values of $Q'$ and a more valuable $Q$-function. So its more desirable to use the advised (and more reliable) actions for a longer period in the initial stages of learning in the target task. This is achieved by the $\epsilon$ - Advice strategy as the target values of $Q$ will need more time to exceed the source $Q'$ values.

The complexity of the proposed algorithm is linear with respect to the in-
stances that are identified in the target task.

5 Experiments

We evaluate the proposed approach in Keepaway. The dimensions of the region where the agents are placed is set to 25m × 25m and remains fixed for all source and target tasks that are considered in the experiments.

The algorithm that is used to train the keepers is the SMDP variation of Sarsa(λ) [10]. Additionally, we use linear tile-coding for function approximation, a method that has been proved effective in several domains, including Keepaway [8]. 32 tilings are used for each variable. The width of each tile is set to 3 meters for the distance variables and 10 degrees for the angle variables. We set the learning rate, \( \alpha \), to 0.125, \( \epsilon \) to 0.01 and \( \lambda \) to 0. These values were selected based on a number of initial experiments without transfer learning and remain fixed for all experiments that are carried out in this work. Algorithm 2 presents the modified Sarsa(\( \lambda \)) algorithm, presented in [8], according to the proposed approach. \( Q' \) is the source function and \( f_S, f_A \) the mapping functions that produced from the decomposition of the target task.

To evaluate the performance of the proposed approach we use the time-to-threshold metric [13], which measures the time required to achieve a predefined performance threshold in the target task. Typically, the threshold is set empirically, after preliminary experimentation in the target task. In our case this threshold corresponds to a number of seconds that keepers maintain ball possession. In order to conclude that the keepers have learned the task successfully, the average performance in 1000 consecutive episodes must be greater than the threshold.

We compare the time-to-threshold without transfer learning against the time-to-threshold with transfer learning plus the training time in the source task.

5.1 4vs3 from 3vs2

We here evaluate the performance of the proposed approach on the 4vs3 target task using a threshold of 9 seconds. We use the 3vs2 task as the source and experiment with different number of training episodes, ranging from 0 (no transfer) to 3200.

Table 4 shows the training time and average performance (in seconds) in the source task, as well as time-to-threshold and total time in the target task for different amount of training episodes in the source task averaged over 10 independent runs. The time-to-threshold without transfer learning is about 14.43 hours. The best time-to-threshold and total time are highlighted with bold typeface.

We first notice that the proposed approach leads to lower time-to-threshold in the target task compared to the standard algorithm that does not use transfer
Table 4: Training time and average performance in the source task, as well as time-to-threshold and total time in the target task for different amount of training episodes in the source task averaged over 10 independent runs.

<table>
<thead>
<tr>
<th>#episodes</th>
<th>3vs2</th>
<th>4vs3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tr. time</td>
<td>perform.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>50</td>
<td>0.05</td>
<td>3.72</td>
</tr>
<tr>
<td>100</td>
<td>0.11</td>
<td>4.38</td>
</tr>
<tr>
<td>200</td>
<td>0.23</td>
<td>4.67</td>
</tr>
<tr>
<td>400</td>
<td>0.74</td>
<td>6.71</td>
</tr>
<tr>
<td>800</td>
<td>1.73</td>
<td>8.52</td>
</tr>
<tr>
<td>1600</td>
<td>5.19</td>
<td>12.20</td>
</tr>
<tr>
<td>3200</td>
<td>13.22</td>
<td>16.84</td>
</tr>
</tbody>
</table>

learning. The more the training episodes in the source task, the less the time-to-threshold in the target task. This is an expected behavior, as the agents’ knowledge, and consequently the advised action, improves with the number of episodes. Note that for 800 episodes of the 3vs2 task, the keepers are able to hold the ball for an average of 8.5 seconds, while for 1600 episodes their performance increases to 12.2 seconds, which shows that they have learned a better $Q$-function.

The total time of the proposed approach in the target task is also less than the time-to-threshold without transfer learning apart from the case, where the agents were trained for 3200 episodes in the source task. The best performance is 9.84 hours, which corresponds to a reduction of 31.8% of the time-to-threshold without transfer learning. This performance is achieved when training the agents for 1600 training episodes in the source task.

Figure 4 depicts the percentage of actions that are selected from the current policy (instead of the advised ones) with respect to the number of episodes in the target task, for the different number of training episodes in the source task. We notice that in the case of 50 and 100 source episodes, the advised actions are selected for a very small time period, after which the agents select the currently learned actions. This is a desirable behavior as the advice from the source task is needed only in the initial stages of learning.

As the training episodes in the source task increase, so do the $Q$-values, forcing the agents to continue following the advised actions. It is interesting to note that in the last two cases (1600 and 3200 episodes), only a small percentage of the target actions are selected, as the $Q$-values of the source tasks are quite large.

5.2 Scaling up to 5vs4 and 6vs5

In order to evaluate the performance of the proposed method in larger and harder problems, we experiment with the 5vs4 and 6vs5 target tasks.
Figure 4: Percentage of actions selected from the currently learned $Q$-function in 4vs3, for different numbers of training episodes in the 3vs2 source task.

For 5vs4 the threshold is set to 8.5 seconds. As source tasks we use the 3vs2 task with 800 and 1600 training episodes and the 4vs3 task with 2000 and 4000 training episodes. Table 5 shows the training times, time-to-threshold and their sum for the different source tasks and number of episodes averaged over 10 independent runs. The best time-to-threshold and total time are highlighted with bold typeface. The time-to-threshold without transfer learning is about 25.23 hours.

<table>
<thead>
<tr>
<th>task</th>
<th>#episodes</th>
<th>tr. time</th>
<th>time-to-thr.</th>
<th>total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3vs2</td>
<td>800</td>
<td>1.73</td>
<td>18.12</td>
<td><strong>19.85</strong></td>
</tr>
<tr>
<td>3vs2</td>
<td>1600</td>
<td>5.19</td>
<td><strong>15.56</strong></td>
<td>20.75</td>
</tr>
<tr>
<td>4vs3</td>
<td>2000</td>
<td>3.12</td>
<td>19.47</td>
<td>22.59</td>
</tr>
<tr>
<td>4vs3</td>
<td>4000</td>
<td>7.52</td>
<td>18.23</td>
<td>23.75</td>
</tr>
</tbody>
</table>

Table 5: Average training times (in hours) for 5vs4. The results are averaged over 10 independent trials. With bold typeface are highlighted the best achieved time in 5vs4 and the best total time.

We first notice that for all the source tasks the total training time is reduced.
It is interesting to note that the best time-to-threshold is achieved for the 3vs2 source tasks using fewer episodes than the 4vs3 tasks. This means that using the knowledge from a much simpler task than the target offers more benefit compared to a source task, which is slightly less complex. In addition, the 3vs2 tasks require less training time, as they are easier than the 4vs3 tasks and a good $Q$-function is achieved earlier.

For the 6vs5 task, the threshold is set to 8 seconds. As source tasks we use the 3vs2 task with 1600 training episodes, the 4vs3 task with 4000 training episodes and the 5vs4 task with 3500 and 8000 episodes. Table 6 shows the results in a similar fashion to the previous Table.

<table>
<thead>
<tr>
<th>source task</th>
<th>#episodes</th>
<th>tr. time</th>
<th>6vs5</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
<td>50.08</td>
<td>50.08</td>
</tr>
<tr>
<td>3vs2</td>
<td>1600</td>
<td>5.19</td>
<td><strong>27.42</strong></td>
<td><strong>32.61</strong></td>
</tr>
<tr>
<td>4vs3</td>
<td>4000</td>
<td>7.52</td>
<td>29.72</td>
<td>37.24</td>
</tr>
<tr>
<td>5vs4</td>
<td>3500</td>
<td>5.17</td>
<td>44.58</td>
<td>49.75</td>
</tr>
<tr>
<td>5vs4</td>
<td>8000</td>
<td>16.08</td>
<td>28.82</td>
<td>44.90</td>
</tr>
</tbody>
</table>

Table 6: Average training times (in hours) for 6vs5. The results are averaged over 10 independent trials. With bold typeface are highlighted the best achieved time in 6vs5 and the best total time.

The time-to-threshold without transfer learning is about 50.08 hours. The best total time is achieved when a 3vs2 task (trained for 1600 episodes) is used as the source task reducing the total time roughly 35%. As in the case of 5vs4 we again notice that when a simpler source task is used, both the time-to-threshold and the total training time decreases. Note that in the case of 5vs4 (3500 episodes) as the source task, the improvement is very small. These results demonstrate that the proposed approach scales well to large problems, especially when a small task is used as the source.

Another interesting fact comes up when we compare the average holding times in the training process of the 4vs3 task (4000 episodes) and the 5vs4 task (8000 episodes). In the first case the average holding time for the last 1000 episodes is 7.8 seconds and for the latter case 7.97 seconds. However, when they are used for transfer learning we notice that they almost have the same performance in the target task with 29.72 and 28.82 hours respectively. This is an indication of the role of decomposition. The 6vs5 game is decomposed to 25 5vs4 instances and to 100 4vs3. In the second case the algorithm searches among a larger number of source instances and it is more likely to get a better advice as more situations are considered in the phase of the advice extraction.

5.3 Usefulness of decomposition

In order to verify the usefulness of decomposing the target task, we compare the proposed approach with a variation that uses only one instance of the source task. More specifically, we use the instance that corresponds to the first $K_s$.
keepers and \( T \) takers as they are ordered increasingly to their distance from the ball. This way we select the agents that are nearest to the learning agent and we shall refer to it as nearest agents (NA) mapping. We must mention here that the NA mapping is similar with the one that is used in [13].

We compare the two methods in the 4vs3 target task using the 3vs2 task as source. Table 7 shows the training times spent in 4vs3 task for different amounts of training episodes in the source task along with the total time. The last column shows the corresponding results of the proposed approach (DEC) for comparison purposes.

<table>
<thead>
<tr>
<th>#episodes</th>
<th>3vs2 tr. time</th>
<th>4vs3-NA total time</th>
<th>4vs3-DEC total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>14.43</td>
<td>14.43</td>
</tr>
<tr>
<td>400</td>
<td>0.05</td>
<td>14.94</td>
<td>12.22</td>
</tr>
<tr>
<td>800</td>
<td>1.73</td>
<td>12.78</td>
<td>11.57</td>
</tr>
<tr>
<td>1600</td>
<td>5.19</td>
<td>18.82</td>
<td>9.84</td>
</tr>
</tbody>
</table>

Table 7: Average training times for 4vs3 of the NA algorithm for different number of training episodes in the source task.

The first observation is that in all cases DEC outperforms NA. An interesting outcome is that NA in two cases (400 and 1600) did not succeeded to reduce the total training time. The results indicate that only an instance is not adequate to provide a good advice and verifies that the decomposition helps to reduce the total training time.

An example that justifies this observation is given in Figure 5. The non-shaded agents are those considered by the NA mapping. In this case, \( K_1 \) can pass either to \( K_2 \) or \( K_3 \). Note that these keepers are blocked by takers \( T_1 \) and \( T_2 \) and ball interception is very likely. However \( K_4 \) is quite open, and a pass to \( K_4 \) would seem the appropriate action in this case. This shows that NA mapping is not always the best mapping in the source task and can lead to inferior advice.

6 Related Work

Singh [6] proposes a method that learns how to compose a set of elemental tasks in order to solve a larger task. This comes in contrast with our approach where we decompose a target task to several instances of a source task. Additionally, in [6] the element tasks must be concatenated in order to solve the composed task which is a different way of improving the training time.

A method that automatically identifies subtasks in a source task proposed in [1]. Using an edge detection technique the method finds subtasks in the value function using some special features, and constructs the corresponding graphs. When the agent faces a new task, it constructs graphs and tries to match them with subgraphs in the source task. The matched subgraphs are used to initialize the corresponding part of the value function in the target task. In our method...
we follow a different approach as we identify the different source tasks that are exist in the target task. Also, we do not break down the source function.

Fernandez and Veloso [2] propose an approach that reuses past policies from solved source tasks to speed up learning in the target task. A restriction of this approach is that the tasks (source and target) must have the same action and state space. Our method allows the state and action spaces to be different between the target and the source task. Additionally, the \( \pi - reuse \) algorithm that is proposed bears similarities with our \( \epsilon - Advice \) strategy only regarding the exploration and exploitation of current and past experience.

Madden and Howley [3] introduce an approach that uses rules built on previous experience, in order to guide the agent in unexplored states that are encountered in the target task. The method evaluated in a simple domain (maze) keeping the state and action space the same for all source and target tasks.

Advice based method have been also proposed in the past [12, 15]. Taylor and Stone proposed a method that uses a list of learned rules from the source task as advices to the target task. The authors introduce three different utilization schemes of the advice. Contrary to the ValueBonus scheme we don’t add a value to the recommended action but instead we check only whether to follow the advice or not. Torrey et al. [15] export rules in first-logic order and translate them into advices.

Taylor et al. [12] propose a method, named Transfer via inter-task mappings, that initializes the weights of the target task with the learned weights of the source task using mapping functions. The method depends strongly on the approximation function that is used in both tasks, as the function that transfers the weights is altered according to the approximation method.

Taylor et al. [11] introduce a method that constructs automatically the mapping functions, using exhaustive search on the set of all possible mappings.
The main disadvantage of this method is that the computational complexity grows exponentially to the number of the state variables and actions.

7 Conclusions and Further Work

In this paper we proposed a novel method for transferring experience in RL agents through the decomposition of the target task in several instances of the source task. We evaluated the efficacy of the method in the Keepaway domain and the results demonstrated that the proposed approach helps to reduce the complete training time in the target tasks concerned in the experiments. Also, we verified that the decomposition of the target task plays an important role to the effectiveness of our method and that it scales well with respect to the the size of the target task.

For further work, we intend to investigate the performance of the proposed approach in tasks that they do not belong to the same domain. Furthermore, another issue for future work is to develop an alternative component that will control the decision of whether to follow or not the advice. This is critical when transferring experience between dissimilar tasks where the Q-values are not directly comparable. Additionally, more schemes for extracting an advice can be incorporated to the $\epsilon$–Advice algorithm. For example, another scheme could be to follow the advised action that is proposed by the majority of the source instances.

Finally, we would like to extend the proposed method in model-based RL algorithms. In this case one could learn a model in the target task and then using the proposed method to obtain several instances of the source model which could be used to provide an advice to the target task.

References


Algorithm 2 The modified Sarsa(λ) algorithm according to the proposed approach for keepaway.

1: procedure RLstartEpisode
2:   for all $a \in A^s$ do
3:     $F_a \leftarrow$ set of tiles for $a, s$
4:     $Q_a \leftarrow \sum_{i \in F_a} \theta(i)$
5:   end for
6:   LastAction $\leftarrow \epsilon$ – Advice(ε, s, $f_S, f_A, Q, Q'$)
7:   LastActionTime $\leftarrow$ CurrentTime
8:   $\overrightarrow{c} \leftarrow 0$
9:   for all $i \in F_{\text{LastAction}}$ do
10:     $e(i) \leftarrow 1$
11:   end for
12: end procedure

13: procedure RLstep
14:   $r \leftarrow \text{CurrentTime} - \text{LastActionTime}$
15:   $\delta \leftarrow r - Q_{\text{LastAction}}$
16:   for all $a \in A^s$ do
17:     $F_a \leftarrow$ set of tiles for $a, s$
18:     $Q_a \leftarrow \sum_{i \in F_a} \theta(i)$
19:   end for
20:   LastAction $\leftarrow \epsilon$ – Advice(ε, s, $f_S, f_A, Q, Q'$)
21:   LastActionTime $\leftarrow$ CurrentTime
22:   $\overrightarrow{\delta} \leftarrow \overrightarrow{\delta} + Q_{\text{LastAction}}$
23:   $\overrightarrow{\theta} \leftarrow \overrightarrow{\theta} + \alpha \overrightarrow{\delta} \overrightarrow{c}$
24:   $Q_{\text{LastAction}} \leftarrow \sum_{i \in F_{\text{LastAction}}} \theta(i)$
25:   $\overrightarrow{c} \leftarrow \lambda \overrightarrow{c}$
26:   if agent acting in state s then
27:     for all $a \in A^s$ where $a \neq \text{LastAction}$ do
28:       for all $i \in F_a$ do
29:         $e(i) \leftarrow 0$
30:       end for
31:     end for
32:     for all $i \in F_{\text{LastAction}}$ do
33:       $e(i) \leftarrow 1$
34:     end for
35:   end if
36: end procedure

37: procedure RLendEpisode
38:   $r \leftarrow \text{CurrentTime} - \text{LastActionTime}$
39:   $\delta \leftarrow r - Q_{\text{LastAction}}$
40:   $\overrightarrow{\theta} \leftarrow \overrightarrow{\theta} + \alpha \overrightarrow{\delta} \overrightarrow{c}$
41: end procedure