Learning to Cooperate with Completely Unknown Teammates

Alexandre Neves[2] and Alberto Sardinha[1,2](✉)

[1] INESC-ID, Lisbon, Portugal
jose.alberto.sardinha@tecnico.ulisboa.pt
[2] Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal

Abstract. A key goal of ad hoc teamwork is to develop a learning agent that cooperates with unknown teams, without resorting to any pre-coordination protocol. Despite a vast number of ad hoc teamwork algorithms in the literature, most of them cannot address the problem of learning to cooperate with a completely unknown team, unless it learns from scratch. This article presents a novel approach that uses transfer learning alongside the state-of-the-art PLASTIC-Policy to adapt to completely unknown teammates quickly. We test our solution within the Half Field Offense simulator with five different teammates. The teammates were designed independently by developers from different countries and at different times. Our empirical evaluation shows that it is advantageous for an ad hoc agent to leverage its past knowledge when adapting to a new team instead of learning how to cooperate with it from scratch.

Keywords: Ad hoc teamwork · Multiagent systems · Reinforcement learning

1 Introduction

As robots become more and more ubiquitous in industrial environments, we also start to see them being deployed in other settings, such as homes [4] and hospitals [14]. In tasks that require cooperation, robots should coordinate to achieve a common goal. However, achieving efficient cooperation may be a complex endeavor when robots come from different origins. One way to address this problem is to endow a robot with the ability to learn *on the fly* how to cooperate by observing their teammates and environment.

Ad hoc teamwork [12] aims to address the problem above and thus design an agent that learns on the fly to adapt to unknown teammates in order to complete

This work was partially supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 (INESC-ID multi-annual funding), the HOTSPOT project, with reference PTDC/CCI-COM/7203/2020, and the RELEvaNT project, with reference PTDC/CCI-COM/5060/2021. In addition, this work was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No. 952215.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
G. Marreiros et al. (Eds.): EPIA 2022, LNAI 13566, pp. 739–750, 2022.
https://doi.org/10.1007/978-3-031-16474-3_60

teamwork tasks. Furthermore, these agents must be robust to changes, such as adapting to new teammates and different environments.

Several algorithms for ad hoc teamwork have been proposed over the past few years. The state-of-the-art methods assume an agent has a library of teammate models and/or tasks and propose algorithms that choose on the fly the most appropriate models/tasks to cooperate with the unknown team (e.g., [9,11], and [10]). Other notable examples are the ad hoc agent in [7], where each task within the library is represented as a fully cooperative matrix game, and AATEAM [2] which has a library of teammate models that are learned with attention networks. Probably the most famous algorithm for ad hoc teamwork is PLASTIC-Policy [1], which also resorts to a library of learned policies and teammate models. However, all these algorithms fail to adapt efficiently when an unknown team is very different from the models/tasks within the library.

The main contribution of this work is to address this gap in the ad hoc teamwork literature by combining a transfer learning strategy and PLASTIC-Policy, whereby we use the parameter sharing strategy. Hence, we create a novel ad hoc agent that can efficiently cooperate with an unknown team that differs significantly from the models/tasks within its library.

We also conducted an empirical evaluation in the Half Field Offense simulation domain [5], a modified version of the RoboCup Soccer Simulation 2D sub-league. The results show that an ad hoc agent can indeed take advantage of PLASTIC-Policy combined with a transfer learning method. In our experiments, the ad hoc agent quickly adapts to unknown teammates, exhibiting close-to-optimal behavior from the start.

2 Background

A Markov decision process (MDP) is a mathematical framework for building sequential decision-making algorithms for agents. Formally, an MDP is a 5-tuple (S, A, p, R, γ), where S is a set of states, A is a set of actions, p is the transition probability function for reaching a state s' given that the previous state was s and the action taken was a, $R : S \times A \rightarrow \mathbb{R}$ is the reward received by the agent upon taking an action a from state s, and γ is the discount factor.

Reinforcement Learning (RL) [13] is a subarea of machine learning that aims to learn an optimal policy of an MDP. The most famous RL method is Q-learning [15]. This RL method learns the action-value function directly, $Q_\pi : S \times A \rightarrow \mathbb{R}$, which corresponds to the discounted reward of taking a given action in a given state then following policy π. Q-learning updates its estimate of the Q-value in the following way: $Q(S_t, A_t) \leftarrow (1-\alpha)Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a))$, where α is the learning rate. Hence, the method interpolates, by a factor of α, the current estimate for a given state-action pair with the reward received by taking the given action in the given state added to the best expected discounted reward possible from the next state onward. With a discrete action space and state space, Q-learning converges to the optimal Q-values (i.e., the Q-values of the optimal policy, $Q_{\pi^*}(s, a)$ or $Q^*(s, a)$) if all actions are sampled a large number of times

in all states. An optimal policy is then derived by simply querying which action maximizes the optimal Q-value for a given state: $\pi^*(s) = \arg\max_a Q^*(s, a)$.

Deep Q-Network (DQN) [8] is a popular RL method for estimating the optimal Q-values with a neural network. An advantage of this approach is that it can handle continuous state spaces in complex environments. DQN approximates the optimal Q-values, Q^*, by using a neural network of parameters θ: $Q(s, a; \theta) \approx Q^*(s, a)$. At each step, the agent adds a transition (s, a, r, s') to a replay memory buffer, from which batches of transitions are sampled in order to optimize the parameters of the network by minimizing the following loss:

$$L(\theta_t) = \mathbb{E}_{s,a,r,s' \sim \rho(.)}\left[(y_t - Q(s, a; \theta_t))^2\right] \text{ where } y_t = r + \gamma \max_{a'} Q(s', a'; \theta_{t-1})$$

Here, y_t is called the temporal difference target, and $y_t - Q$ is called the temporal difference error. ρ represents the behavior distribution, the distribution over transitions (s, a, r, s') collected from the environment. The input to a DQN is a state, s, where each feature of the state space corresponds to an input node, and the output are the various values of $Q(s, a)$ where each output node corresponds to a different action a.

DQNs also resort to *experience replay* to make their updates more stable, by storing each transition (s, a, r, s') in a circular buffer called *the replay buffer* at every time step. Then, instead of using a single transition to compute the loss and back-propagate, a mini-batch of past transitions (randomly sampled) is used instead. This improves the stability of the updates by using uncorrelated transitions in a batch and is called batch learning.

3 OTLPP - Online Transfer Learning for Plastic Policy

In this section, we present our algorithm OTLPP that builds on PLASTIC-Policy [1]. The key aim of this algorithm is to learn to collaborate quickly with unknown teammates that are significantly different from previously encountered teams.

PLASTIC (Planning and Learning to Adapt Swiftly to Teammates to Improve Cooperation) is an algorithm described in [1]. When an ad hoc agent uses PLASTIC, it observes how its team acts and models that behavior to predict the optimal action to take. PLASTIC-Policy is the policy-based version of PLASTIC, which allows it to work in complex and continuous state space domains.

PLASTIC-Policy maintains a probability distribution over all team policies within its library, representing how likely the current team is to its library's team model/policies. The ad hoc agent builds this library from the previous interaction with teammates. To update the distribution, the agent observes the team while it acts and determines which past team is most likely to be the current one. A limitation of PLASTIC-Policy is that it is only effective on a team that resembles a team within its library, meaning the current team must be similar to one of the previously encountered teams so that it can cooperate adequately.

```
 1: function LearnAboutPriorTeammate(t)                                  ▷ t: the prior teammate
 2:    DQN ← a newly initialized Deep Q-Network
 3:    Data ← ∅
 4:    s ← the initial state
 5:    repeat
 6:       a ← DQN(s)
 7:       Take action a and collect (s, a, r, s′)
 8:       Data ← Data ∪{(s, a, r, s′)}
 9:       Add (s, a, r, s′) to DQN's replay buffer and perform batch learning on DQN
10:       s ← s′
11:    until no more transitions are needed
12:    Derive a policy π for t from DQN
13:    Learn a nearest neighbors model m of t using Data
14:    return (π, m)
15: end function
16:
17: function UpdateBeliefs(BehaviorDistr, s, s′)     ▷ BehaviorDistr: probability distribution over possible
    teammate behaviors, s: the previous environment state, s′: the new environment state
18:    for (π, m) ∈ BehaviorDistr do
19:       loss ← 1 − P(s′|m, s)
20:       BehaviorDistr(m) ←BehaviorDistr(m)∗(1 − ηloss)
21:    end for
22:    Normalize BehaviorDistr
23:    return BehaviorDistr
24: end function
25:
26: function SelectAction(BehaviorDistr, s) ▷ BehaviorDistr: probability distribution over possible teammate
    behaviors, s: the current environment state
27:    (π, m) = argmax BehaviorDistr                                  ▷ select the most likely policy
28:    return π(s)                          ▷ return the action for the given state according to that policy
29: end function
```

Fig. 1. Pseudocode for the policy based implementation of the methods `LearnAboutPriorTeammate`, `UpdateBeliefs`, and `SelectAction` (methods taken from [1] and adapted to use a DQN).

To solve this problem, Sect. 3.2 presents an algorithm that combines a transfer learning algorithm with PLASTIC-Policy, which is the key contribution of our work. In Sect. 3.1, we present the methods from PLASTIC-Policy that have been adapted to use a DQN.

3.1 PLASTIC-Policy Methods

PLASTIC-Policy relies on the `LearnAboutPriorTeammates` function in Fig. 1 to build a team model and policy regarding a previously encountered team. The agent gathers data in the form of the tuple (s, a, r, s')[1] as it plays with a given team. With these tuples, it obtains a policy by using a Deep Q-Network, which is an RL algorithm that uses samples from the transitions to approximate the values of the state-action pairs. It also learns a nearest neighbors model from the same gathered data, to be used later by the PLASTIC-Policy algorithm when making predictions about the new team's behavior.

In `LearnAboutPriorTeammates`, a Deep Q-Network is initialized in Line 2 and the algorithm queries the DQN for which action it should take next (Line 6) and executes it, collecting the tuple (s, a, r, s') in Line 7. Line 10 adds the tuple to `Data` so that the tuples can be used to build the nearest neighbors model in Line 13. In Line 9, the tuple is also added to the DQN's replay buffer, and the DQN then proceeds to perform batch learning, as described in Sect. 2. Once no more transitions are deemed necessary, the agent derives a policy from the Deep

[1] Where s is the original state, a the action taken, r the reward received and s' the resulting state.

```
1: function TransferKnowledge(BehaviorDistr)  ▷ BehaviorDistr: probability distribution over all known teams
   regarding which one is most similar to the current one.
2:    (π, m) ← argmax BehaviorDistr                                              ▷ get the best policy
3:    DQN_source ← the Deep Q-Network associated with π
4:    DQN_target ← a new Deep Q-Network
5:    Use Parameter Sharing between DQN_source and DQN_target
6:    return DQN_target
7: end function
```

Fig. 2. Pseudocode for the transfer learning algorithm used in this article.

Q-Network (Line 12), which can be done by simply saving a snapshot of the network's weights and later using `DQN`(s) to select an action, just like Line 6.

PLASTIC-Policy keeps a probability distribution over all previously encountered teams within its library. These beliefs are updated at each timestep as the agent gathers more data regarding the current team. The function `UpdateBeliefs`, in Line 17 of Fig. 1, presents the pseudocode for updating the beliefs. Line 18 iterates through all team models (i.e., nearest neighbors models) so that, in line 19, it can use the model, along with the original state s, to calculate the probability that the resulting state s' is consistent with the current team. This is done by getting $\hat{s} = m(s)$, where $\hat{s}$ is the closest state in m to s, for some distance measure[2]. Then, the corresponding resulting state, $\hat{s'}$ is obtained from $\hat{s}$[3]. Furthermore, each state feature is compared between s' and $\hat{s'}$ to calculate the probability, where the difference between the two is due to a noise drawn from a normal distribution. These probabilities are then multiplied together to obtain $P(s'|m, s)$. In line 20, the probability distribution is updated for the current team model by a factor of $1-\eta$loss. The η factor is used to attenuate sporadically incorrect predictions that would otherwise bring the probability of the potentially correct team close to 0. Once all beliefs have been updated, the distribution is normalized (Line 22).

While the agent updates its beliefs, it must also act in the environment to gather further data about the current team. Therefore, it must use one of the policies at its disposal to do so. Choosing which policy to use comes down to simply choosing the team that has the highest belief, according to what the agent has seen so far, as shown in function `SelectAction` (Line 26 of Fig. 1).

3.2 Combining Transfer Learning and PLASTIC-Policy

Online Transfer Learning for PLASTIC-Policy (OTLPP) is the key contribution of this work, where the ad hoc agent can collaborate with a new team that is significantly different from the team models and policies within its library. OTLPP can effectively leverage the ad hoc agent's past experiences with different teams to quickly adapt to a new one by combining PLASTIC-Policy with Transfer Learning.

[2] The only requirement for the distance measure is it to be 0 when $\hat{s} = s$ and close to 0 when $\hat{s}$ and s are considered "similar".

[3] Such association was recorded in `LearnAboutPriorTeammates` (Fig. 1, line 10).

```
1: function OTLPP(PriorTeammates, HandCodedKnowledge, BehaviorPrior)                          ▷
   PriorTeammates: past teammates the agent has encountered, HandCodedKnowledge: prior knowledge coded
   by hand, BehaviorPrior: prior distribution over the prior knowledge.
2:
3:   PriorKnowledge ← HandCodedKnowledge                        ▷ initialize knowledge from prior teammates
4:   for t ∈ PriorTeammates do
5:     PriorKnowledge ← PriorKnowledge ∪ LearnAboutPriorTeammate(t)
6:   end for
7:   BehaviorDistr ← BehaviorPrior                                                ▷ initialize beliefs
8:
9:   Initialize s
10:  repeat                                           ▷ see which team is the most similar to the current one
11:      a ← SelectAction(BehaviorDistr, s)
12:      Take action a and observe r, s′
13:      BehaviorDistr = UpdateBeliefs(BehaviorDistr, s, s′)
14:      s ← s′
15:  until one team has high enough probability
16:
17:  DQN ← TransferKnowledge(BehaviorDistr)               ▷ use Transfer Learning to adapt to the new team
18:
19:  Initialize s once again
20:  repeat                              ▷ begin learning with the jump start from the transferred knowledge
21:      a ← DQN(s)
22:      Take action a and collect (s, a, r, s′)
23:      Add (s, a, r, s′) to DQN's replay buffer and perform batch learning on DQN
24:      s ← s′
25:  until the agent has learned
26: end function
```

Fig. 3. Pseudocode for OTLPP algorithm, which combines transfer learning with the PLASTIC-Policy algorithm from [1].

One common technique for transferring knowledge between two neural networks is to share the parameters (or part of them) between the source network and the target network [16]. This technique is known as Parameter Sharing, and we used it to transfer knowledge between DQNs. There are several strategies in Parameter Sharing, such as transferring the weights of all layers or just the weights from the shallower (closer to the input) layers. Also, the shallower weights may be frozen, which implies learning will only occur in the deeper layers[4].

The transfer learning algorithm, represented by the function `Transfer Knowledge` in Fig. 2, begins by obtaining the policy that corresponds to the team with the highest probability value, in Line 2. Then, in the following line, it derives a Deep Q-Network from that policy[5], which will be used as the source for the transfer learning algorithm. The target is a newly initialized Deep Q-Network. The final step is to use Parameter Sharing from the source DQN to the target DQN. As mentioned above, all or just a subset of the weights may be transferred, and weight freezing may occur.

In Fig. 3, we present the OTLPP algorithm, which combines transfer learning with PLASTIC-Policy. In Lines 3–7, and much like the original PLASTIC-Policy algorithm, the agent begins by combining some knowledge that may have been hand-coded with knowledge gathered from previously encountered teams by using the `LearnAboutPriorTeammate` function (see Fig. 1). At this point, the ad hoc agent has compiled a library of policies and team models. Therefore, it can reason about the similarities between the current team and each of these past teams.

[4] This is done for learning stability reasons.

[5] The policy is actually computed via a Deep Q-Network, so this derivation is straightforward.

In Lines 9–15, the ad hoc agent continuously updates its behavior distribution over the past teams with `UpdateBeliefs` (Fig. 1), while also selecting the next action to take based on the current values of the distribution with `SelectAction` (Fig. 1). Once a certain team reaches some probability threshold of acceptability, indicating that it is indeed the right team, the agent can move on to transferring knowledge (Line 17) from its prior experiences and begin learning a new policy for the new team. In other words, Line 17, obtains a Deep Q-Network whose parameters have been initialized by `TransferKnowledge` (Fig. 2). It can then begin to learn the new policy for the new team, in lines 19–25, with the advantage of not having to start learning from scratch due to the Transfer Learning algorithm.

4 Experimental Setup

This section describes our experimental setup to test the OTLPP algorithm within an agent. We used the Half Field Offense (HFO) simulator [5], a modified version of the RoboCup Soccer Simulation 2D sub-league. In HFO, an offense team tries to score a goal and the defense team tries to prevent it. In addition, only half of the original field is playable, and a defending agent cannot attack nor vice-versa. In following paragraphs, we present the experimental setup in detail.

Test Setting and Agents: All tests were conducted in a 2 vs 2 matches in the HFO, where the ad hoc agent plays with one teammate against two opponents. The opponents are two instances of the `agent2d` and the teammate can be one of the following: `agent2d`, `aut`, `gliders`, `helios`, from 2013, and `receptivity`, from the 2019 RoboCup Soccer Simulation 2D sub-league competition[6] What this means is that the opponents will always act according to the `agent2d` strategy, whereas the teammate can display one of 5 behaviors, according to each of the 5 different types of teammates mentioned above. No behavioral variability was allowed for the opponents, since *ad hoc teamwork* is not concerned with changes in the task (which includes besting the opponents), hence their singular strategy.

State Space: HFO provides two state spaces on the fly: a low-level one and a high-level one. The low-level state space provides $59 + 9T + 9O$ features, where T is the number of teammates (excluding the agent) and O is the number of opponents. These features include the positions and velocities of all agents on the field and of the ball, the sines and cosines of various angles (e.g., goal opening, agent orientation, velocity vector orientation), and other game state variables. The high-level state space provides a higher-level view of the match, which combines the lower-level features into more meaningful ones, exposing only $12 + 6T + 3O$ features. Of these, the following 12 features were selected as the state space for our agent: X position (the agent's X-position on the field); Y position (the agent's Y-position on the field); orientation; ball X (the ball's X-position on the field); ball Y (the ball's Y-position on the field); goal opening angle; proximity to opponent; teammate's goal opening Angle; proximity from the teammate to opponent; pass

[6] Agent binaries were downloaded from the following page: https://archive.robocup.info/Soccer/Simulation/2D/binaries/RoboCup/.

opening angle; teammate's X (the X-position of the teammate; teammate's Y (the Y-position of the teammate). We included the agent's position, along with its orientation and the ball's position, for obvious reasons: the agent cannot make the simplest of decisions if it does not know its position and the ball's location. The goal opening angle feature is present so that the agent may decide whether it is worth shooting toward the goal or pass to a teammate, among other scenarios. This is also why we include the proximity to the opponent, the teammate's goal opening and pass angles, and the teammate's position.

Action Space: Although HFO offers action spaces of different levels of abstraction. We decided to use a new action space that resorted almost entirely to delegating to the high-level actions already provided by HFO and preventing some unintended behavior, as explained below. As a result, the following 11 actions were made available to the agent: shoot; short-dribble; long-dribble; pass-to; no-op (the agent takes no action for 4 time-steps); go-to-ball; go-to-goal; go-to-teammate; go-away-from-teammate; go-to-nearest-opponent; go-away-from-opponent.

Reward Function: To create a reward function, we use the status of the game after an action has been taken. The following statuses are available to the agent: in-game (the game is still ongoing); goal; captured-by-defense; out-of-bounds; out-of-time; server-Down. Hence, we use PLASTIC-Policy's reward function [1], which also used the HFO to evaluate the agent's performance. The reward function is the following:

$$R(status) = \begin{cases} 1000 & \text{if } status \text{ is } \texttt{Goal} \\ -1 & \text{if } status \text{ is } \texttt{In-Game} \\ -1000 & \text{otherwise} \end{cases}$$

Deep Q-Network Parameters: The ad hoc agent was first trained using a Deep Q-Network against all 5 different teammate types from scratch. This allowed the agent to have a library of approximately optimal policies for each teammate type. The network has an input layer of 12 nodes to accommodate each of the 12 features from the state space, 3 hidden layers of 512 nodes, an output layer of 11 nodes to indicate the estimated q-value of each of the 11 actions, Rectified linear unit activation functions between all layers, a learning rate of 0.00025, a replay memory capacity of 2.5×10^5 transitions (beginning its use when it has at least 12500 transitions stored), a learning batch size of 64 transitions, an ϵ-greedy action selection (with ϵ having a linear decay that begins at 0.8 and decays to 0.05 after 100000 time-steps), a discount factor, γ, of 0.995, an Adam optimizer [6], a transfer rate of 500 time-steps, and a weight initialization sampled uniformly from $[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$, where n is the amount of nodes in a given layer.

OTLPP Parameters: Like PLASTIC-Policy, our algorithm relies on a parameter, η, to attenuate sporadic losses on the correct team model. Also, the noise distribution used to compute $P(s'|m, s)$ is a normal distribution of mean 0 and variance σ^2, so σ is also another parameter. The following values were

used in our code: $\eta = 0.10$, and $\sigma = 4.0$. The condition `one team has high enough probability` in Line 15 of Fig. 3 is true at the end of the 25th game, given that at this stage, the agent almost always makes the correct decision regarding its current team. In the `UpdateBeliefs` function (Fig. 1), the probability $P(s'|m, s)$ is calculated by first obtaining the state $\hat{s}$ that is closest to s via the nearest-neighbor model m, obtaining that state's corresponding next state, $\hat{s}'$, and then calculating the distance between the next state and the predicted next state, $d = \texttt{Distance}(s', \hat{s}')$. `Distance` can be defined in several different ways, but fundamentally it should be 0 if $\hat{s}' = s'$ and close to 0 if $\hat{s}'$ and s' are considered similar. For this work, and since the intent is to identify which teammate the agent is playing with, the features of the state space that do not pertain to the teammate, such as the opponents' movements or the agent's own movements, should not be considered. Therefore, the only features considered are the teammate's coordinates. Hence, let s_x be the teammate's x-position in a given state s, and s_y its y-position analog. Then, `Distance` is defined as $\texttt{Distance}(s', \hat{s}') = \prod_{c \in \{x,y\}} \texttt{ProbFromNoise}(s'_c - \hat{s}'_c)$ and $\texttt{ProbFromNoise}(\Delta) = 1 - 2 \cdot |F_{N(0,\sigma^2)}(\Delta) - \frac{1}{2}|$, where $F_{N(0,\sigma^2)}$ denotes the cumulative distribution function for a normal distribution of mean $\mu = 0$ and a variance of σ^2. Therefore, `ProbFromNoise` returns the probability that a given value was drawn from a normal distribution. As mentioned before, we use a normal distribution in the equation above because we assume that every transition is affected by the noise of the normal distribution.

Parameter Sharing: During the Transfer Learning stage of the OTLPP algorithm (Fig. 3), the agent uses Parameter Sharing to transfer knowledge between a single source Deep Q-Network and the target Deep Q-Network that will later learn the new policy. We decided to transfer all weights, including those from deeper layers, because the environment and the task remain the same. Therefore, a lot of commonality is expected between how the ad hoc agent should behave when paired with the old teammate and with the new one. However, if the change in teams represents a higher change in the overall problem, perhaps transferring only the shallower layers, which correspond to broader abstractions, could also be used.

5 Results

In our empirical evaluation, the task consists of having our ad hoc agent (with the OTLPP algorithm) cooperate with one of 5 different teammates, namely (`agent2d`, `aut`, `gliders`, `helios`, and `receptivity`). Each teammate was designed independently by developers from different countries for the 2013 and 2019 RoboCup Soccer Simulation 2D sub-league competition. Furthermore, the goal is to score as many goals as possible against a defense team made up of two agents of the type `agent2d`.

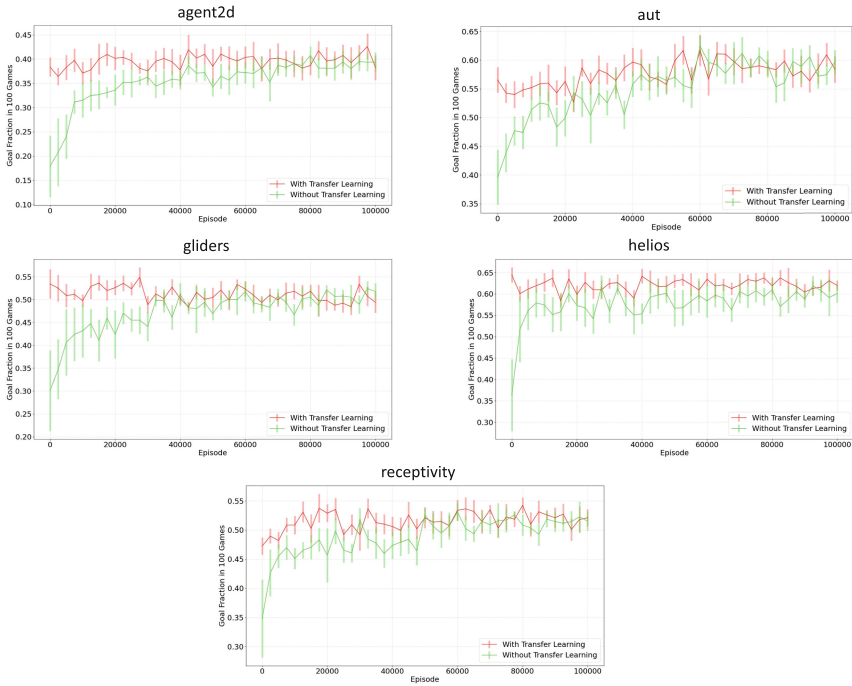

Fig. 4. Comparison between the average goal fraction when the agent uses OTLPP (in red) compared to an RL agent (in green), while playing with each teammate. The vertical bars represent one standard deviation. (Color figure online)

Given that our agent uses a DQN, one way to evaluate the learning procedure at a given point in time is to take a snapshot of its network's parameters at that time and use them to obtain the goal fraction (i.e., the number of goals divided by the number of games). In our experiments, all goal fractions were obtained by running 200 games for the same network parameters and dividing the number of goals scored by that amount. Other metrics may be used, such as the sum of the rewards or the average time to score. However, since the task is ultimately to score a goal, the goal fraction is probably the best metric.

To test the algorithm against each teammate available, we excluded the policy of the teammate from the library, which is the equivalent to calling `LearnAboutPriorTeammates` (Fig. 1) on all teammates except for the particular one during the execution of OTLPP (Fig. 3). Hence, the ad hoc agent learns the policy of 4 teammates and then uses that knowledge, along with the behavior distribution it maintains over them, to transfer knowledge to a new Deep Q-Network for the 5th teammate.

In Fig. 4, we start showing the ad hoc agent's performance during the learning process, when it resorts to the OTLPP algorithm (red line) with each teammate: `agent2d`, `aut`, `gliders`, `helios`, and `receptivity`. We also include the performance of an RL agent (green line), which depicts the performance of an

Table 1. Comparison between the average goal fraction over 100000 episodes when the agent uses PLASTIC-Policy and when it uses OTLPP, for each team.

Team	PLASTIC-Policy	OTLPP
`agent2d`	0.37	**0.40**
`helios`	0.61	**0.62**
`aut`	0.55	**0.58**
`gliders`	**0.53**	0.51
`receptivity`	0.47	**0.52**

agent that learns from scratch. From the plots, we can notice that our ad hoc agent with the OTLPP algorithm benefits from a significant boost, when transfer learning takes place at the beginning of learning. In other words, our agent is able to exhibit close-to-optimal behavior from the start.

Finally, Table 1 compares OTLPP's performance with PLASTIC-Policy during run time. In particular, Table 1 compares the average score of an ad hoc agent using PLASTIC-Policy with an ad hoc agent using OTLPP, over 100000 episodes. Each ad hoc agent has 4 policies within its library, and we exclude the policy that corresponds to the current teammate of the ad hoc agent. This evaluation enables us to test the performance of the ad hoc agent with a completely unknown teammate. In bold, we highlight the maximum score between both alternatives. The results show that OTLPP can outperform or perform close to PLASTIC-Policy in every test, showcasing the advantage of our OTLPP algorithm. Hence, OTLPP almost always means an improvement over PLASTIC-Policy.

6 Concluding Remarks

Our paper is the first work to combine a transfer learning method with PLASTIC-Policy in order to create an ad hoc agent that learns to cooperate with a completely unknown teammate. In our empirical evaluation, we show that the combination of a transfer learning method with PLASTIC-Policy can serve as a powerful tool for ad hoc agents. In particular, the results show that OTLPP can outperform or perform close to PLASTIC-Policy. Hence, the ad hoc agent quickly adapts to completely unknown teammates, exhibiting close-to-optimal behavior from the start. As future work, other transfer learning methods may be explored, such as transferring knowledge from multiple sources [3], meaning multiple teams would be a source of transferred knowledge.

References

1. Barrett, S., Rosenfeld, A., Kraus, S., Stone, P.: Making friends on the fly: cooperating with new teammates. Artif. Intell. **242**, 132–171 (2017)
2. Chen, S., Andrejczuk, E., Cao, Z., Zhang, J.: AATEAM: achieving the ad hoc teamwork by employing the attention mechanism. In: AAAI Conference on Artificial Intelligence, vol. 34, pp. 7095–7102 (2020)

3. Ge, L., Gao, J., Zhang, A.: OMS-TL: a framework of online multiple source transfer learning. In: 22nd ACM International Conference on Information and Knowledge Management, pp. 2423–2428 (2013)
4. Iocchi, L., Holz, D., del Solar, J.R., Sugiura, K., van der Zant, T.: RoboCup@Home: analysis and results of evolving competitions for domestic and service robots. Artif. Intell. **229**, 258–281 (2015)
5. Kalyanakrishnan, S., Liu, Y., Stone, P.: Half field offense in RoboCup soccer: a multiagent reinforcement learning case study. In: Lakemeyer, G., Sklar, E., Sorrenti, D.G., Takahashi, T. (eds.) RoboCup 2006. LNCS (LNAI), vol. 4434, pp. 72–85. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74024-7_7
6. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization (2014). https://arxiv.org/abs/1412.6980
7. Melo, F.S., Sardinha, A.: Ad hoc teamwork by learning teammates' task. Auton. Agent. Multi-Agent Syst. **30**(2), 175–219 (2016)
8. Mnih, V., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
9. Ribeiro, J.G., Faria, M., Sardinha, A., Melo, F.S.: Helping people on the fly: ad hoc teamwork for human-robot teams. In: Marreiros, G., Melo, F.S., Lau, N., Lopes Cardoso, H., Reis, L.P. (eds.) EPIA 2021. LNCS (LNAI), vol. 12981, pp. 635–647. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86230-5_50
10. Ribeiro, J.G., Martinho, C., Sardinha, A., Melo, F.S.: Assisting unknown teammates in unknown tasks: ad hoc teamwork under partial observability (2022). https://arxiv.org/abs/2201.03538
11. Santos, P.M., Ribeiro, J.G., Sardinha, A., Melo, F.S.: Ad hoc teamwork in the presence of non-stationary teammates. In: Marreiros, G., Melo, F.S., Lau, N., Lopes Cardoso, H., Reis, L.P. (eds.) EPIA 2021. LNCS (LNAI), vol. 12981, pp. 648–660. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86230-5_51
12. Stone, P., Kaminka, G.A., Kraus, S., Rosenschein, J.S.: Ad hoc autonomous agent teams: collaboration without pre-coordination. In: Twenty-Fourth AAAI Conference on Artificial Intelligence, pp. 1504–1509 (2010)
13. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, 2nd edn. MIT Press, Cambridge (2018)
14. Tasaki, R., Kitazaki, M., Miura, J., Terashima, K.: Prototype design of medical round supporting robot "Terapio". In: 2015 IEEE International Conference on Robotics and Automation (ICRA), pp. 829–834 (2015)
15. Watkins, C.J.C.H., Dayan, P.: Q-learning. Mach. Learn. **8**, 279–292 (1992)
16. Zhuang, F., et al.: A comprehensive survey on transfer learning. Proc. IEEE **109**(1), 43–76 (2020)