Progress Report for CS: 449-549 - Learning for Robotics
Analysis of Collaborative Learning Through Different MARL Algorithms in PressurePlate

Kaan Ünlü
Computer Science
Bilkent University
Ankara, Turkey
kaan.unlu@bilkent.edu.tr

Mert Kaan Er
Mathematics
Bilkent University
Ankara, Turkey
m.er@ug.bilkent.edu.tr

Alper Bozkurt
Computer Science
Bilkent University
Ankara, Turkey
alper.bozkurt@ug.bilkent.edu.tr

I. Introduction

Reinforcement Learning (RL) propels artificial intelligence into collaborative territories with multi-agent Reinforcement Learning (MARL). The PressurePlate environment, a cooperative multi-agent benchmark, presents a unique challenge.We implemented several modifications and changed the default layout so that the problem is more challenging than the original. Agents navigate a grid-world, triggering doors through designated pressure plates. The goal is to reach a specific zone for a successful episode. Our exploration involves employing MARL algorithms to enhance agent collaboration in this distinctive context.

II. Pressureplate Environment and modifications

A. The original Pressureplate environment

The pressureplate environment is a cooperative multi-agent benchmark environment created for testing multi-agent reinforcement learning (MARL) algorithms[1] [4]. The environment is a grid world. In the environment, there are four zones sectioned by walls. Each zone is connected to the next zone via doors, which are triggered when a pressure plate is pressed by a specific agent. Each door is assigned a single pressure plate, and each pressure plate can be triggered by a specific agent. If the respective agent steps on a pressure plate, the associated door opens and stays open as long as the agent stays on the plate. The moment agents move from the plate, the door is closed. At the end of the three zones, there is the goal zone, and the episode is successful when an agent reaches the goal. The optimal behavior for agents is to step on the pressure plate that they are assigned(if they are assigned one) as soon as possible and stay on the plate until the end of the episode. The agent that cannot activate any pressure plate must reach the goal as soon as possible. Figure 1 displays the snapshot of the initial state, and Figure 2 displays the goal completion state of the original pressure plate environment.

[1]https://github.com/uoe-agents/pressureplate

Each agent has five discrete action choices for each time step: move up, move down, move right, move left, or stay. The transition dynamics are deterministic.

Each agent has its own partial observation and doesn't have access to either the global state of the environment or the observation of other agents. The partial observations consist of the absolute coordinates of the agent and a flattened five-layer grid centered around the agent, whose size is defined by the sensor range(environment parameter). The layers of the grid identify the present tile type around the agent at a certain position. The tile types are agents, walls, doors, plates, and goal. Assuming the sensor range is m, the observation of an agent is an array of length $5 * m^2 + 2$.

The pressureplate repository [4] neither contains any default MARL algorithm nor suggests the use of any specific algorithm.

B. Modifications to Pressureplate environment

We decided that the original problem was quite easy for modern MARL algorithms to solve and doesn't require a complex form of cooperation from the agents. We implemented various modifications to the base environment to make the problem more challenging.

We were not satisfied with the expected optimal behavior for an agent being moved to a pressure plate and waiting there until the end of the episode. We implemented a timing mechanism for plates and doors. In our implementation, once a suitable agent presses a plate, the door associated with the plate stays open for a fixed time step whether the agent moves away from the plate or not. This mechanism allows agents to leave their plates and interact with the environment after pressing their plates. With this paradigm, all of the agents are now able to reach the goal. Hence, the only sensible design

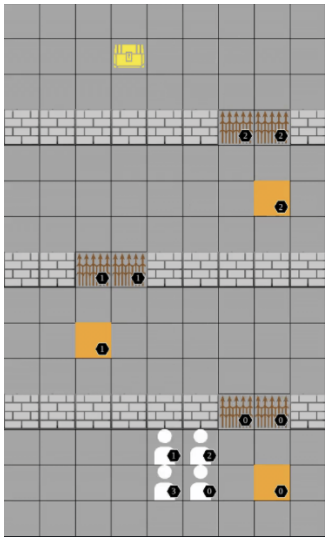

Fig. 1. The initial state of the original pressureplate environment.

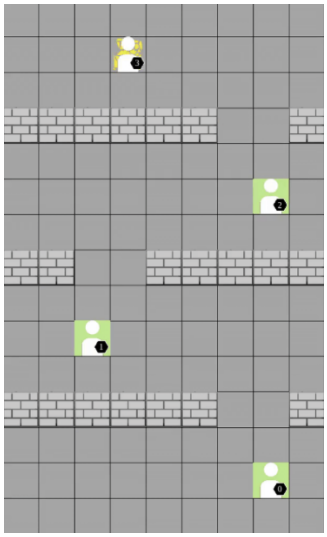

Fig. 2. The goal completion state of the original pressureplate environment.

choice for goal completion is that all of the agents reach the goal zone instead of one of the agents reaching the goal as in the original implementation. Moreover, in our version, multiple designated agents can interact with pressure plates, but each agent is able to keep the door open for a certain period of time. This paradigm forces agents to discover which of them is best suited to use the plate among the ones that can interact with the plate so that the episode can be completed in the shortest possible period of time.

For these changes to formulate a meaningful and challenging task, we changed the layout of the environment. Figure 3 and Figure 4 displays two snapshots of the layout we designed. The grid size is 28 by 14, and we used four agents. There are two types of plates associated with specific doors. Agent-0 and Agent-1 can activate Plate-1 for 5 turns and 12 turns, respectively, while Agent-2 and Agent-3 can activate Plate-2 for 17 turns and 5 turns, respectively, to give agents some small room for mistakes in an effort to make the learning less over-optimised/overfit to the environment.

We preserved the partial observation constraint that came with the original implementation so that the agents don't overfit the layout; instead, they make their decisions based on the local information that they have access to, which enables them to adopt the new layouts faster. However, we slightly changed the observation representation. In the original observation representation, five layers were used to describe the five different possible tile types. We decided that a single layer can be set up to contain the same information. Reducing the number of layers to 1 significantly decreased the observation space dimension. Also, we added to the observations that the duration for which the agent can keep the door open. In our implementation, observation space consists of a single-layer flattened grid of shape senor range by sensor range, absolute coordinates of the agent, and the activation duration. Therefore, given that the sensor range is m, observation of a single agent is a m^2+3 dimensional array.

Coming up with a reward function is the most challenging part of creating a reinforcement learning(RL) benchmark environment. An RL agent not necessarily learns to complete the task, but it learns to maximize the reward. Hence, the reward function must be set up in such a way that maximization of it leads to the completion of the task in the shortest possible time. One way of defining a reward function is to give a positive reward to agents upon completion of the task. Theoretically, such a reward function would lead to agents learning the correct behavior; however, practically, since the rewards are too sparse, agents wouldn't learn to behave in a reasonable training time. Therefore, implementing the reward function based on sub-goals(such as reaching a certain zone or pressing a plate) is paramount to solving the problem in a reasonable training time. We designed the reward function, therefore, with the following constraints in mind:

- Agents should receive a ticking time penalty per every motion where they have not arrived at their final goal, the treasure, scaling with their distance to the treasure, to encourage them to advance forward.
- Agents should be incurred with checkpoint penalties based on whether they reach a certain distance up to the goal by a set amount of timesteps, to discourage dawdling behaviour.
- The agents should be rewarded with small positive rewards depending on the action, but said rewards should never tick up the per action reward up to positive, to stop endless repeats of said actions. These rewards can be divvied up as such:
  - Agent should receive reward from how many agents are in reward zone. It is okay for this reward to tick up the per turn reward up to the positives.

– Agents should receive reward from how many other agents have passed a door while they are standing guard at a door, to discourage them from letting the door close on their teammates.
– Agents should receive reward from passing a door before their key countdown expires after they leave a pressure plate, again to discourage them from dawdling on pressure plates.

Certain malfunctions within the code unfortunately still withhold the reward function from working correctly, as in longer stretches (past the agents' sensor range), the pathfinding to the next door falters massively, blocking progress completely.

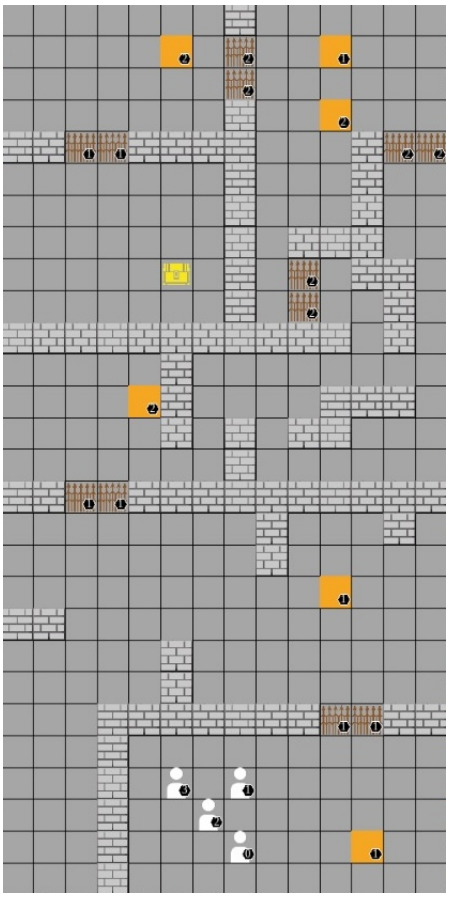

Fig. 3. The initial state of our implementation.

III. Multi Agent Reinforcement Learning

MARL algorithms are generally built upon single-agent reinforcement learning algorithms. An algorithm designed for single-agent environments can be implemented for multi-agent environments without any modifications. From the perspective of a single agent, other agents can be formulated as a part of the dynamics of the environment, and for each agent, the environment can be formulated as a single-agent environment. However, such a formulation would be sub-optimal. The policy of each agent is changing over time, and hence, from the perspective of a single agent, the dynamics of the environment

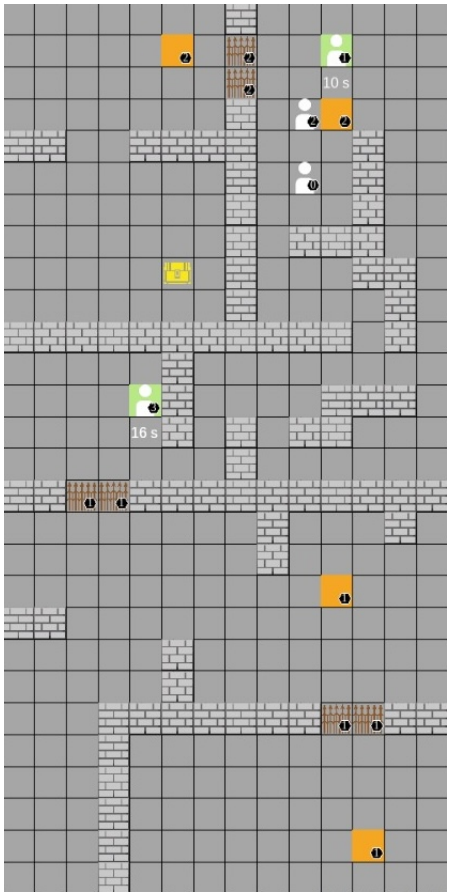

Fig. 4. An intermediate state of out implementation.

are changing in an unpredictable way. This problem leads to unstable learning. One of the proposed ways of solving this problem is incorporating the observation of other agents during training but relying only on the observations of individual agents during evaluation [2].

A. Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient(DDPG) algorithm is an off-policy actor-critic algorithm designed for learning the optimal policy in a single-agent continuous action space environment [1]. In the DDPG algorithm, (s, a, r, s') pairs are stored in a replay buffer to be used for learning the actor and critic functions. The actor $\mu(s|\theta^\mu)$ is a deterministic policy parameterized by a neural network, and the critic $Q(s, a|\theta^Q)$ is the state-action value parameterized by another neural network. Also, the target networks μ' and Q' are maintained for generating the target values for the critic. The parameters of target networks are updated by the formula $\theta' = \tau\theta + (1-\tau)\theta'$ where τ is a hyperparameter.
Given a batch of N samples $\{(s_i, a_i, r_i, s'_i)\}|_{i=1}^{i=N}$ the target values calculated as $y_i = r_i + \gamma Q'(s'_i, \mu'(s'_i|\theta^{\mu'})|\theta^{Q'})$ where γ is the discount factor. The critic $Q(s, a|\theta^Q)$ is optimized to

minimize the loss

$$L = \frac{1}{N}\sum_{i=0}^{i=N}(y_i - Q(s_i, a_i|\theta^Q))^2$$

. The the parameters of the actor function $\mu(s|\theta^\mu)$ are updated with the policy gradient

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_{i=1}^{i=N} \nabla_a Q(s_i, a = \mu(s_i|\theta^\mu)|\theta^Q)\nabla_{\theta^\mu}\mu(s_i|\theta^\mu)$$

and then the target networks are updated using exponentially weighted moving average as described above.

B. Multi Agent Deep Deterministic Policy Gradient

Multi Agent Deep Deterministic Policy Gradient(MADDPG) modifies the DDPG algorithm so that it can be effectively used in multi-agent algorithms [2]. MADDPG algorithm maintains an actor and critic for each agent. It uses centralized training and decentralized execution. In other words, the critic function of each agent has access to the observations and actions of every agent, while the actor functions only have access to the observation of the agent they are paired with. With this formulation via critic functions, actors have information about the beliefs of other agents about state-action pairs, and every agent can learn what other agents are planning to do. Hence, this formulation assumes agents can communicate during training. However, since the actor functions have only access to the observation of a single agent during execution(or evaluation), communication between agents is not assumed at this stage, making this formulation suitable for environments in which agents don't communicate. Allowing agents to communicate during training leads to more stable learning, and it enables agents to recognize other agents as, in fact, independent agents.

Even if each critic function has the same input, their respective target values are calculated using their agent's reward. Therefore, the use of a separate critic function for each agent is necessary. A possible workaround for using a single critic could be setting up a common reward function in which each agent receives the same reward not based on their individual actions but their cumulative actions. However, the use of such a reward function could lead to some agents being lazy as they receive rewards for the actions of other agents [3]. Consequently, as we decided not to rely only on a global reward function, we had to use one critic function per agent. Perhaps with an overhaul of the reward function this might become viable in the future.

MADDPG inherently only works for environments with continuous action spaces as it requires the $Q(s, a)$ to be a differentiable function with respect to the action. Fortunately, there are several gradient estimation techniques for estimating the gradient of discrete functions. These techniques are explored and integrated into MADDPG by Tilbury et al. [5]. We used the Temperature-Annealed Gumbel Softmax function to estimate the gradient of the critic as described in the paper.

IV. Experiments and Results

In conclusion, our modifications to the PressurePlate environment have transformed it into a more challenging and dynamic testing bed for MARL algorithms. The implemented changes encourage cooperative strategies among agents, pushing them to adapt and optimize their behaviors for efficient goal completion. Leveraging the MADDPG algorithm, we navigate the complexities of multi-agent collaboration, fostering stable learning and effective communication during training. The reward function needs further work to make the learning be functional in the complexity we have envisioned to test it in. As we delve into the subsequent sections, we will explore the details of our implementation, the intricacies of more MARL algorithms, and a reach a more sterilized and well processed set outcomes of our experiments, aiming to contribute valuable insights to the field of cooperative multi-agent environments in RL.

References

[1] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Yuval Tassa Tom Erez, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2015.

[2] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments, 2017.

[3] Hangyu Mao, Zhibo Gong, and Zhen Xiao. Reward design in cooperative multi-agent reinforcement learning for packet routing, 2018.

[4] Trevor McInroe and Filippos Christianos. Pressureplate.

[5] Callum Rhys Tilbury, Filippos Christianos, and Stefano V. Albrecht. Revisiting the gumbel-softmax in maddpg, 2023.