Overcooked-AI a fully cooperative task

Daniel Lopes 93696
Eduardo Miranda 95569
Rodrigo Pinto 95666

ABSTRACT

Overcooked-AI is a benchmark environment designed to test the performance of fully cooperative task coordination, based on the popular video game Overcooked. In this environment, agents must work together to deliver soups as fast as possible by coordinating tasks and distributing them amongst each other. The game requires effective communication, planning, and decision-making, making it a challenging test of cooperation. Different agent architectures will be used on this fully cooperative coordination task and each one will be evaluated according to metrics in order to see which one provides the best results. All the members of the group worked together to achieve the pretended goal.

KEYWORDS

Overcooked-AI; Cooperation; Coordination; MARL; Architectures

1 INTRODUCTION

Multi-agent reinforcement learning (MARL) is becoming increasingly important as more AI systems are being deployed. Many potential applications of MARL involve dynamic interactions between agents, such as agents adapting to each other, ad-hoc coordination, and more[1].

1.1 Motivation

The main motivation behind this project subject is to learn more about coordination and cooperation in a multi-agent system in a visual way. To do this, a fast paced cooperative environment that allows this (Overcooked-AI) will be studied and used.

1.2 Related Work

The Overcooked-AI environment is mentioned and introduced on the article "On the Utility of Learning about Humans for Humans-AI Coordination" [5]. Despite being that within the article the authors use the environment to learn about Human-AI coordination, an algorithm for coordination will be developed with the usage of the environment instead. Multi-agent reinforcement learning (MARL) is an important topic in order to be able to understand the function of Human-AI, this topic further explored within The Dynamics of Reinforcement Learning in Cooperative Multi-agent Systems [2]. The article focused primarily on Q-learning in cooperative multi-agent systems under the perspective that the agents explicitly attempt to learn the value of joint actions and the strategies of their counterparts, focusing on the influence of that game structure and exploration strategies on convergence to (optimal and sub-optimal) Nash equilibria. Within Reinforcement Learning of Coordination in Cooperative Multi-agent Systems different action selection strategy for Q-learning improving on the results from the previous papers mentioned [4]. These two papers will be the highlight focus of the study in order to develop an algorithm using a reinforcement learning architecture that maximizes the shared utility between the agents in the task at hand.

1.3 Problem definition

Overcooked-AI is an environment based on the video game called Overcooked [5]. The environment consists in various "maps" where each "map" is a different kitchen. There are two chefs (agents) and both have the same goal that is to deliver the maximum amount of soups within a specified time limit. Each soup requires placing up to three ingredients in a pot, waiting for the soup to cook, and then having an agent pick up the soup and delivering it. The agents should distribute tasks and coordinate effectively in order to achieve high reward. This is a Markov Decision Process problem because it has full observability and actuation.

1.4 Objectives

The objective is to find which type of agent architecture is the most suitable for the Overcooked-AI environment. Alongside, there is also a goal to discover patterns and interesting behaviours that the agents may develop to achieve the goal.

2 APPROACH

2.1 Environment Description

Being that there are 2 chefs functioning (2 agents), a **multi-agent** environment is the bases for this in order for them to be able achieve the goal of delivering soups as fast as possible.
This environment is **accessible** since both agents can obtain complete, accurate, up-to-date data about the environment state. The agents can "see" the full kitchen at each time step allowing for this to function.
Each action in this environment has a single guaranteed effect therefore the environment is **deterministic**. The environment is also **dynamic** because when one of the agents is deliberating the other is moving and performing tasks.
The environment is **discrete** because it has a fixed and finite amount of possible actions.
Finally the environment is **episodic**, it is possible to divide the world in series of intervals independent of each other. In this specific environment an episode is a time limited run in one of the "kitchens".

2.2 Architecture Description

2.2.1 *Decision-making capabilities*

. **Cooperation** is needed because the agents have the same goal.

AASMA Project, IST, 2023, Lisbon, Portugal

https://doi.org/

They also need **reinforcement learning** in order to discover the best way to coordinate between one another to maximize rewards. Two RL algorithms will be implemented, the first one will be **SARSA with a random policy** and the other will be **Q-Learning**.

2.2.2 Sensors: one of the key components that enables agents to interact with and comprehend their environment
. In this environment each agent has 5 sensors:

- **players** (list(PlayerState)): A list of the currently active players, where each element of a list is a PlayerState.
- **objects** (dict(tuple:list(ObjectState))): A dictionary mapping positions (x, y) to ObjectStates, but it does NOT include objects held by players (they are in the PlayerState objects).
- **bonus-orders** (list(dict)): Current orders worth a bonus.
- **all-orders** (list(dict)): Current orders allowed at all.
- **timestep** (int): The current timestep of the state.

Besides these 5 sensors there are auxiliary states:

- **ObjectState** A dictionary with the name of the object and its position.
- **SoupState** Represents a soup object. An object becomes a soup the instant it is placed in a pot. It stores the position (tuple), ingredients(list(ObjectState)), cooking(integer representing how long the soup has been cooking) and cooktime(integer representing how long the soup needs to be cooked) of the soup.
- **PlayerState** Stores the position (tuple), orientation (NORTH, SOUTH, EAST, WEST) and held-object (ObjectState).

2.2.3 Actuators: are the mechanism through which these agents are capable of exerting physical control, enabling them to interact, manipulate, and influence their environment in meaningful ways.

- **NORTH:** moves the agent upwards
- **SOUTH:** moves the agent downwards
- **EAST:** moves the agent to the right
- **WEST:** moves the agent to the left
- **STAY:** the agent remains in the same place
- **INTERACT:** the agent can pickup and drop ingredients, plates and can turn on the pot

2.2.4 Why are these sensors and actuators useful?
Sensors are needed to have a view of the current map layout and also to know what the other agent is doing for cooperation purposes. Actuators are the possible actions that the agents can take to achieve the goal.

2.2.5 Architectures we are going to use
. Two types of Reinforcement Learning architecture will be used to represent the decision making capabilities. One of them will be **SARSA with a random policy**, where each agent chooses their actions randomly. This approach will be used as the baseline as a comparison to the other approach. The other approach will be **Q-Learning** where the qvalues of each agent will be updated every time they perform an action until it reaches the optimal qvalue function. This approach was opted for instead of a deliberative or an Hybrid one because using a RL architecture with enough training steps the agents will learn the optimal way of doing soups.

2.3 Evaluation Methods

One qualitative metric will be used to evaluate the performance of our approaches, **Episode Reward**. This metric takes into account not only the goal that is to deliver soups, but also sub-goals like putting ingredients in the pot, picking up dishes or soups. In depth, if one agent delivers a soup, both agents receive a +20 reward; if an agent puts an ingredient in the pot, only that agent will receive a +3 reward; if an agent makes a useful dish pick up (a dish pick up is defined as useful if there is a soup ready in the pot) it will receive +3 reward; if an agent picks up a soup it will receive a +5 reward.

3 SOLUTION

Python implementation was built where each agent architecture is represented by a class containing the respective characteristics and behaviors. This allowed for a simulator over the Overcooked-AI environment to be built using these architectures to do experimental simulations and evaluate agent performance in multiple scenarios.

3.1 Configurable Parameters

The implementation supports, different configuration parameters. These parameters can be modified in the train.yaml file. The parameters that can be changed include: the map in which the agents are going to be trained, the number of steps per episode, the number of total training steps, the evaluation frequency which indicates the number of training steps between each evaluation, the number of evaluation episodes in each evaluation, a boolean called **common reward** that controls if a soup delivery rewards both agents or only one, the initial exploration probability, the exploration probability decay and the minimum exploration probability. Besides this variables we also have variables related to how frequently the training logs and if a video is saved. It is also possible to modify the learning rate and the gamma value of the Q-learning and SARSA algorithms in their respective .yaml files.

3.2 Agent Training

In order to guide and help with the implementation some inspiration was taken from "Offline Multi-Agent Reinforcement Learning Implementations: Solving Overcooked Game with Data-Driven Method", regarding training offline agents in the Overcooked-AI environment[3].

3.2.1 External loop
. To train the agents a loop must be made until the number of training steps defined is reached in the configuration parameters. In the loop if it is the start of a new episode, the environment must reset and updates the exploration probability. After, the agents act according to their current observations and with the actions they choose, a step in the environment is ran that gives us the next observation and rewards. With this information an update occurs in each agent. This way of coding allows for reusability since agent classes can be created that encapsulate the behaviour of different reinforcement learning algorithms.

3.2.2 How a new training algorithm is created?
To create a new agent following a new algorithm, a new class with

the name of the algorithm (example: **q-learning.py**) needs to be created, and this class will have a list with agents and different parameters, which allows agents that follow different reinforcement learning techniques in the same map. The class also defines what act and update methods do. Each agent inside that class is an instance of a different class (ex- ample: **QLAgent.py**) that defines the methods of how the agent acts and updates itself according to the rewards and observations. These two classes should be in files inside the "model" folder. Finally, we should define configuration files for the new algorithm that we created (example: **qlearning.yaml**).

3.3 SARSA

Within the experiment, the baseline would be SARSA with the random policy. State–action–reward–state–action (SARSA) is an algorithm for learning a Markov decision process policy. A SARSA agent interacts with the environment and updates the policy based on actions taken, hence this is known as an on-policy learning algorithm. SARSA updates its q values using the following equation:

$$Q_{new}(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad (1)$$

Since SARSA with the random policy will result in random exploration it is a good baseline to have.

3.4 Q-learning

We decided to use Q-learning to compare with our baseline since Q-learning learns the optimal q values function, being an **off-policy** algorithm.
An off-policy algorithm is an algorithm that uses different policies for acting and updating. As we see in the update, the policy used to act is not used:

$$Q_{new}(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma * max_{a'}(Q(s_{t+1}, a')) - Q(s_t, a_t)] \quad (2)$$

3.5 Exploration vs Exploitation

In both algorithms, it is necessary to balance exploration and exploitation, since in the beginning the agents do not have any knowledge about the environment so they should prioritize taking random actions to explore new things about the environment. Once the agents start to get more knowledge, they should gradually prioritize exploitation, i.e taking the best action. This helps to avoid convergence to a local minimum for the Q-values function, which most of the time is far from the global minimum. To handle this we use a threshold that decays every episode using the exponential decay formula:

$$N(t) = N_0 e^{-\lambda t} \quad (3)$$

N_0 is the initial value and λ is the decay constant.
At every time step t, we sample a variable uniformly over [0,1]. If the variable is smaller than the threshold, the agent will explore the environment. Otherwise, he will exploit his knowledge.

4 RESULTS

Training episodes and evaluation graphs are compared within the results. Training episodes show the rewards obtained by the agents in each training episode. Each data point in the evaluation graphs show the average rewards of three evaluation episodes (evaluation episodes are episodes that do not update the agents) at that training step. However evaluation frequency and other parameters can be configured.

4.1 Default Parameters

To simplify experimental analysis, the following default parameters for each parameter type in our system was used.

env	"cramped room"
episode_length	500
num_train_steps	40000
eval_frequency	5000
num_eval_episodes	3
common_reward	true
exploration_prob	1
exploration_decreasing_decay	0.01
min_exploration_prob	0.01
lr	0.01
gamma	0.95

4.2 Baseline (SARSA with random policy)

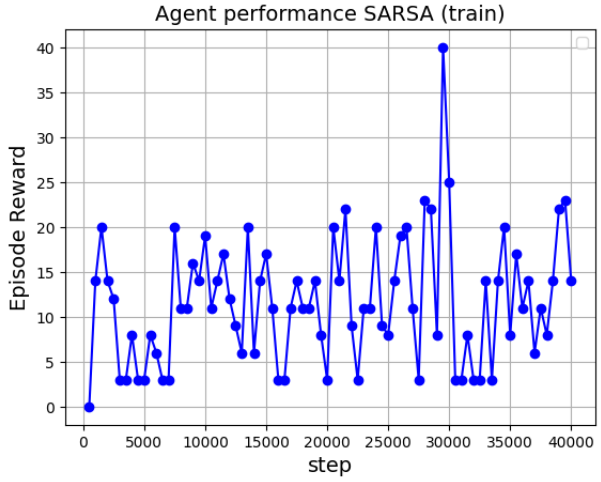

Figure 1: Rewards obtained with SARSA by both agents in each training episode

From figure 1 it is evident that SARSA with a random policy isn't ideal because the overall trend of the episode reward remains constant throughout the training, meaning that the agents are not learning much. This becomes clearer in training step 29500 where the agents obtained a lot of rewards by delivering a soup and doing a lot of sub-goals, however in the next training episodes the agents show no consistency in delivering the soup, confirming that the agents did not learn how to obtain those rewards confirming our assumption from the solution section that SARSA with the random policy just results in random exploration.

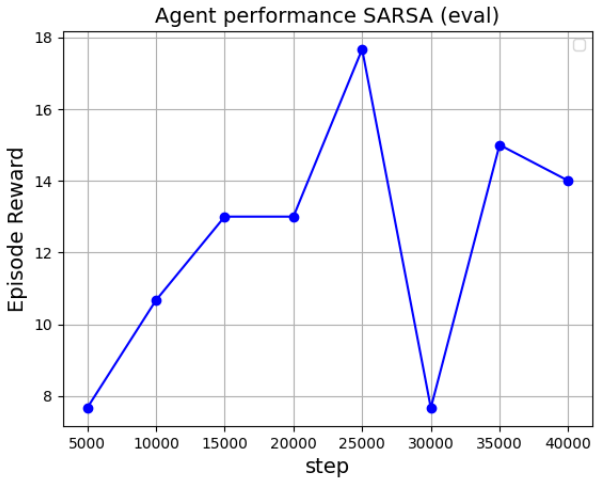

Figure 2: Rewards obtained with SARSA by both agents in each evaluation episode

From the evaluation episodes in figure 2 there is no clear trend. This is specially clear from the comparison of the evaluations at step 25000 to 30000 where the rewards dropped from 17.667 to 7.667. It is also important to note that at the last evaluation ϵ has already decayed to 0.45 meaning that by this evaluation the agents already have a higher probability of exploiting instead of exploring, however as we can observe from both figures 1 and 2 the agents seem to no be learning, this corroborates what we concluded from figure 1 that SARSA using the random policy will just result in random exploration.

4.3 Q-Learning

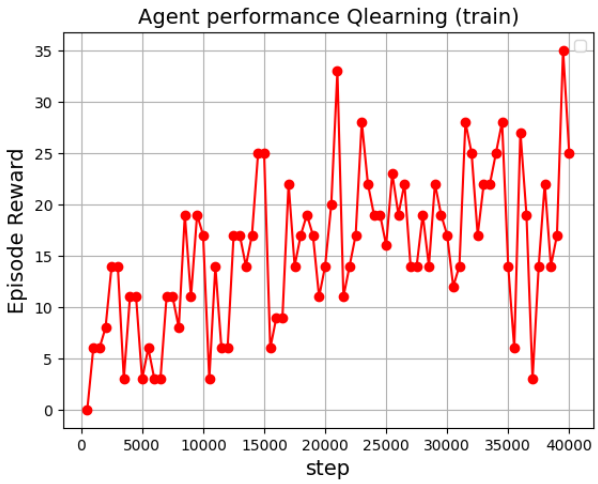

Figure 3: Rewards obtained with Q-Learning by both agents in each training episode

From figure 3 it can be observed that the graph has an upwards trend indicating that the agents are learning how to maximize rewards. Unfortunately, due to hardware limitations, we could not run Q- Learning enough time for it to converge to the optimal solution. However, it is already possible to see that from roughly the training step 20000 the agents are already consistently delivering soups except for two outliers on training steps 35500 and 37000. In these two runs the agents chose more exploration actions than exploitation since ϵ had a value of 0.49 and 0.48 respectively so they still had a high probability of choosing exploring actions but, in these two runs in particular, this did not pay off.

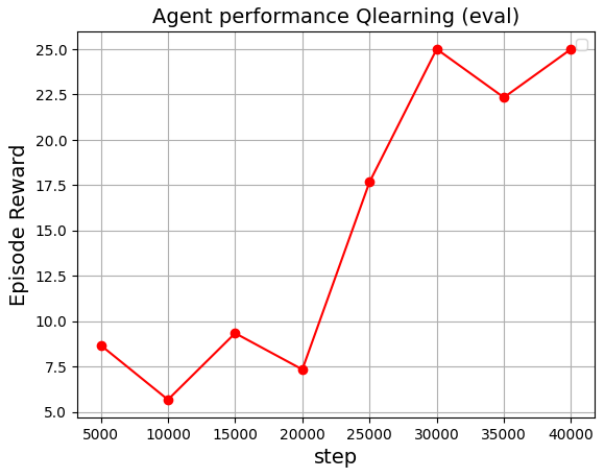

Figure 4: Rewards obtained with Q-Learning by both agents in each evaluation episode

The agents performance on the evaluation episodes corroborate what we concluded from figure 3 in the training episodes. Here the trend is upwards and we can see that from the point the agents made the first soup they consistently made it again in the next evaluations (from evaluation in training step 25000 to 40000). Unfortunately, as explained above, the algorithm was unable to run for more training steps, due to a lower decay rate and more training episodes the agents could try more random actions and discover how to make the second soup, third soup, etc until the algorithm converged to the optimal solution that would be the maximum amount of soups in an episode. But, despite this, the agents can make the first soup consistently showing clear signs of learning.

4.4 SARSA vs Q-Learning

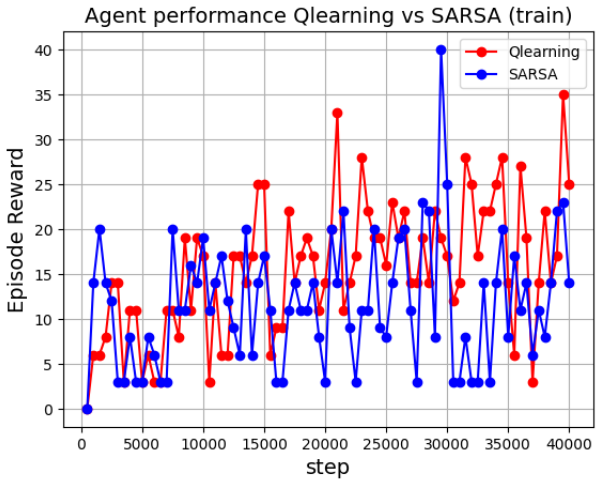

Figure 5: Rewards obtained with Q-Learning vs rewards obtain with SARSA by both agents in each training episode

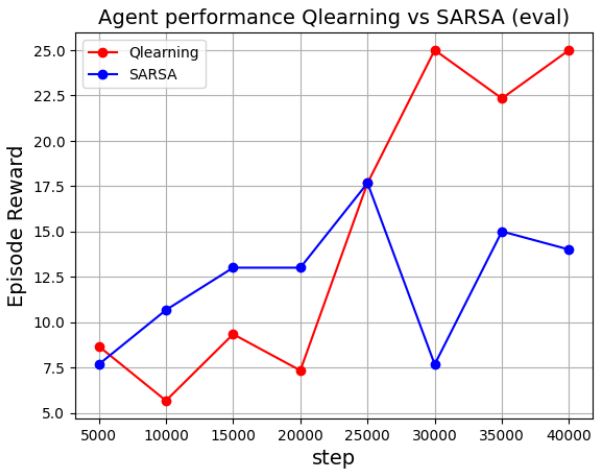

Figure 6: Rewards obtained with Q-Learning vs rewards obtained with SARSA by both agents in each evaluation episode

From these comparisons, it is prevalent that Q-Learning achieves better results than SARSA with the random policy, this is specially clear in figure 6 on the evaluations comparison. This difference in results may be due to the fact that Q-Learning is 'off-policy', learning the Q-Values for the greedy policy and SARSA is 'on-policy' using the same policy to act and update its value function. This means that SARSA is learning the Q-values for the random policy which as we may deduce is not the best policy for our problem, because it ends up resulting in just random exploration and limited learning. While Q-Learning is learning the optimal q-values which results in more rewards. Despite there being insufficient number of training iterations, it is likely that this difference in rewards would only increase since the trend for Q-Learning is upwards and for SARSA is constant.

5 CONCLUSIONS

During the experimental simulations, the impact of different reinforcement learning algorithms on the amount of rewards obtain during an episode was evaluated. From the two algorithms Q-Learning was the one that achieved better results, however Q-Learning did not run for enough time to converge to the optimal solution because the environment is complex and therefore the number of states is large.

5.1 Ideal configuration

Ideally, there should be a ϵ decay slowly and a low learning rate so that Q-values function converged to the optimal solution. Since if the ϵ decays too quickly the agents would not have tried enough random actions to find the optimal solution and if the learning rate is too large the Q-values function will never converge to the actual optimal solution because the steps are too big. So if we were not limited by hardware the parameters that could be possibly used include:

env	"cramped room"
episode_length	500
num_train_steps	2350000
eval_frequency	5000
num_eval_episodes	3
common_reward	true
exploration_prob	1
exploration_decreasing_decay	0.001
min_exploration_prob	0.01
lr	0.001
gamma	0.95

The number of training steps would be 2350000 because this would give 4700 episodes and if this value is inputted and the decay rate in the decay formula it gives an exploration probability of 0.009 so there is no need of more episodes.

5.2 Other interesting algorithms

In this experiment, two of the most known reinforcement learning algorithms were used. However it is important to note that there are techniques that were not use that could have helped reduce the state space which could optimize Q-Learning taking less time to train. Some of these techniques include: Function Approximation, Experience Replay and State Abstraction.

The random policy for SARSA was used, however this policy could be changed to a policy that would have produce better results for SARSA like the soft-max policy. There are also some reinforcement learning algorithms that can match or even surpass the performance of Q-Learning and SARSA, due to faster convergence. Some of the algorithms that are worth to explore are Deep Q-Learning, Proximal Policy Optimization and Soft Actor-Critic. Besides reinforcement learning algorithms, it is also worth exploring different agent architectures that use coordination approaches like social conventions and roles.

REFERENCES

[1] Andy Shih Dorsa Sadigh Bidipta Sarkar, Aditi Talati. 2022. PantheonRL: A MARL Library for Dynamic Training Interactions. (2022). https://iliad.stanford.edu/pdfs/publications/sarkar2022pantheonrl.pdf

[2] Caroline Claus and Craig Boutilier. 1998. The Dynamics of Reinforcement Learning in Cooperative Multiagent Systems. (1998). https://cdn.aaai.org/AAAI/1998/AAAI98-106.pdf

[3] Baek In-Chang. 2021. Offline Multi-Agent Reinforcement Learning Implementations: Solving Overcooked Game with Data-Driven Method. (2021). https://github.com/bic4907/Overcooked-AI

[4] Spiros Kapetanakis and Daniel Kudenko. 2002. Reinforcement Learning of Coordination in Cooperative Multi-agent Systems. (2002). https://cdn.aaai.org/AAAI/2002/AAAI02-050.pdf

[5] Mark K. Ho Thomas L. Griffiths Sanjit A. Seshia Pieter Abbeel Micah Carroll, Rohin Shah and Anca Dragan. 2020. On the Utility of Learning about Humans for Human-AI Coordination. (2020). https://arxiv.org/pdf/1910.05789.pdf