Multi-agent Game Play

1st Yuan Yang
yyang998@gatech.edu
Deep Reinforcement Learning

***Abstract*—The experiments presented in this work are based on the Overcooked-AI environment [1], which mimics a multiplayer game where two players make and deliver soups in a kitchen layout. This project extends my Deep Q-Learning (DQN) implementation in Project 2 by developing Double DQN (DDQN) [1] on top of it and further adjusting the implementation to solve the multi-agent gameplay problem. To better address this Overcooked multi-agent environment, techniques such as reward shaping, weight sharing, and reward sharing are implemented to enhance performance. Overall, four algorithms are developed in this work: IndependentDQN, CentralizedDQN, IndependentDDQN, and CentralizedDDQN. Among these algorithms, Centralized algorithms are designed to better handle multi-agent scenarios.**
¡

I. Introduction

The Overcooked-AI environment [1] is designed based on the video game Overcooked, where two players optimally cook and deliver onion soups. Four off-policy reinforcement learning algorithms were implemented with improvements. The main focus is on enhancing the DQN algorithm to Double DQN [1] and modifying the design to better address this environment, which requires two agents to act both individually and collaboratively. Experience Replay has also been employed.

This paper begins by introducing the Overcooked environment and the kitchen layouts used in this work, followed by an explanation of the implemented algorithms and the justification for the implementation decisions based on the environment's characteristics. The key point in this work is to handle the multi-agent case using various strategies. The performance of training and testing is then evaluated and compared across all five layouts. The effects of different design considerations and hyperparameter choices are also discussed. Additionally, several additional metrics are presented to better explain the events during gameplay that can result in better soup deliveries. Moreover, some pitfalls encountered and avenues for future work to enhance performance are also discussed.

II. Overcooked Environment [1]

In the Overcooked environment [1], two agents play together to cook onion soups, each containing three onions, and successfully deliver the soups. This task involves multiple subtasks: picking up onions at onion dispenser, placing onions in a pot located at a specific location in the kitchen, waiting for the soup to cook, picking up a dish plate, placing the soup in the dish after completion, and delivering the soup to the designated serving area. These subtasks make the environment complex, and each step has its impact on whether a soup can be successfully cooked and delivered. Having two players in the same kitchen working on the same task further escalates the difficulties in task completion.

Moreover, to optimally play the game and deliver as many soups as possible within the restricted amount of time (400 timesteps) in the environment, the effectiveness of the agents is crucial. Therefore, various events and states of each agent during the process could have a significant impact on the outcome. Examples of such events and states include (not limited to):

- whether any player stays still and does nothing within a certain amount of time;
- whether players dropped any onions/plates/soup;
- whether players put more than the necessary amount of onions into the pot (more than 3 onions);
- distance from the agents' current location to the target places (pot/onion/plate/serving areas);
- trajectory that the agents walked from one location to another and if this path is optimal, etc.

Most importantly, if two agents collaborate to deliver soups by performing each subtask respectively for cooking one soup, then the effectiveness could be potentially improved to a large extent.

During the gameplay, the agent can take actions in a discrete action space that contains six possible actions: up, down, left, right, stay put, and interact. The interact action will be triggered by the agent's certain state (e.g., interacting to put an onion in the pot when facing the pot location and holding an onion in the agent's hands).

For the state space, this is a Markov decision process, and each agent can access the full present observation within the environment. The features of each agent include position information, whether the agent is holding any object (onion/plate/soup), distance from this agent to different objects (onion/plate dispenser, pot, serving area), distance from one agent to another, etc.

For the rewarding mechanism, each soup delivery generates 20 points as a reward. As mentioned earlier, given the complexity of this system and the varied impact of sub-events on successful soup delivery, different sub-events should be rewarded differently to facilitate effective Q-learning. Additionally, it is important to differentiate the rewards for different features (states) of an agent at the current time step ((this is my future work).

There are five kitchen layouts that need to be solved: cramped room, asymmetric advantages, coordination ring,

forced coordination, and counter circuit 0 1order. Successfully solving a layout requires delivering an average of at least 7 soups in the recent 100 episodes. All five layouts are treated with a single algorithm using the same setup and hyperparameter values.

III. Algorithms and Strategies Explanation

A. DQN and DoubleDQN

In this work, my first step is to upgrade the DQN implemented in Project 2 to Double DQN [2], as outlined in Project 2's future work. As introduced in my Project 2 report, standard DQN [3] [4] uses a single network Q for both action selection and action evaluation in the Q-learning update. This introduces maximum bias since it estimates the maximum values using a single network during evaluation. After bootstrapping, this bias can be significantly elavated, inducing potential problems and leading to unstable and slow training.

For Double DQN, two networks are designed to handle action selection and estimation separately. The estimation is completed using two networks: the primary network and the target network. When computing the target Q value during learning, the target network Q' is used for action selection, while the primary network Q is used for action evaluation at each time step:

$$Q^*(s_t, a_t) \approx [r_t + \gamma Q(s_{t+1}, argmax_{a'} Q'(s_t, a_t))] \quad (1)$$

During learning, in order to mitigate the biases, the weights of target network is constantly copied from the weights of primary network using Polyak averaging:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \quad (2)$$

where θ' is the weight parameters of target network, θ is the weight parameters of primary network, τ is the rate of averaging.

To implement DDQN, these are the major modifications I made based on my project 2's DQN algorithm:

- Create two networks with same depth and width.
- During learning, compute target Q values based on Eq1.
- During learning, update target network's weights based on Eq2.
- Use primary network in environment steps when using ϵ-greedy to get action.

Please refer to Algorithm 1 in project 2 report for details steps of DQN. The setup of the experience replay buffer in DDQN is the same as DQN implementation.

B. Independent DQN and DoubleDQN with Reward Sharing

In order to address this multi-player environment, I initially implemented two independent DQN/DDQN agents to represent the two players, without any cooperation between them. Each agent has its own target Q-network, primary Q-network, and replay buffer. In this setup, each agent only adds its own experience to its respective replay buffer, and one agent cannot use another agent's experience for network learning.

To introduce a slight extent of cooperation, the reward used by each agent is the global reward, which represents the number of soups delivered at the current time step by both agents together.

The performance of this algorithm is not ideal, and the results are presented in the Results section below. To further improve this algorithm and better address cooperation, Centralized DQN/DDQN is designed and introduced in the next subsection.

C. Centralized DQN and DoubleDQN for Multi-Agent Purpose - Parameter Sharing

Next, I designed centralized DQN/DDQN based on the independent ones. Some multi-agent RL ideas are adapted from previous works. Foerster et al. [5] introduced the idea of parameter sharing, where various agents in the system can share the same parameters for learning and training. This is essentially similar to ideas introduced in various multi-agent reinforcement learning works for centralized-training-with-decentralized-execution methods [6] [7] [8].

The basic idea is that various agents share a network. In some actor-critic approaches [7], multiple agents share one actor and one critic during training, which is the so-called centralized training. What I did in this project is to adapt this idea to DoubleDQN and have two agents in the system share one target network, one primary network, and the same replay buffer.

Detailed modifications include: During training, each agent saves their experiences into the same replay buffer, samples from the same replay buffer, and uses the same Q-network for learning (parameter sharing). Therefore, each agent can learn from other agents' past experiences and better optimize its own strategy. Moreover, for two agents, the two actions from each of them together result in the outcome reward; doing centralized training could potentially be better for the realization of this coordination.

The performance was improved on certain layouts, and the results are discussed in the Results section. However, for the layout that forces absolute cooperation between agents by separating them into different rooms, the performance is not ideal.

Furthermore, I designed a reward shaping system to couple with Centralized DoubleDQN to help agents better learn and promote or suppress the effect of different events during training.

D. Reward Shaping and Sharing

During reward shaping, as shown in Table 1, I assigned different rewards or penalties to various events that occur at each timestep. For instance, if the agent drops an ingredient/object at the current time, it receives a penalty in its reward for that timestep. Conversely, if the agent picks up a dish plate/soup/onion, particularly if it's a useful one, the agent receives an additional reward added to its total reward.

I set the reward and penalty values in reward shaping to be relatively small compared to the reward for soup delivery (+20). The goal is to ensure that the reward for soup delivery remains the dominant factor, as it is the ultimate goal to

maximize. If the effect of certain event overshadows soup delivery, then learning direction would be distracted. However, we still need to differentiate the effects of events to aid in achieving better soup delivery outcomes. Some events represent important sub-steps in successfully delivering soups, while others are harmful, and some events have minimal impacts.

The implementation details is: at the end of each timestep, the status of the game is obtained by calling

```
base_env.game_stats
```

. For each of the two agents, if the current time step number is included in the list under a certain event, the shaped reward for this agent will receive the corresponding reward/penalty. The shaped reward is then accumulated throughout the current episode. The implementation details are in reward_shaping.py.

TABLE I: Reward Shaping

Event	Value	Event	Value
onion pick	0	useful dish drop	-1
useful onion pick	0.5	soup pick	2
onion drop	-0.5	soup delivery	2
useful onion drop	-0.5	soup drop	-2
potting onion	0.5	optimal onion pot	0.5
dish pick	0.2	viable onion pot	0.5
useful dish pick	1	catastrophic onion pot	-1
dish drop	-0.5	useless onion pot	-1

E. *Other Strategies*

To achieve better training performance, some additional strategies are also coupled with main approach. To help in more stable training:

- During ϵ-greedy, the ϵ value is decayed at the end of every episode instead of decaying after every time step. This approach ensures more stable training. In the early episodes, the model can focus on exploring the environment extensively to prevent missing any optimal values. As the episodes progress, the model can then shift its focus to learning from the most optimal scenarios based on previous experiences.
- The gradient clipping strategy for the neural network [9] is adapted to further stabilize the training process. This approach helps prevent very large updates to network weights and, in turn, avoids overshooting/missing the optimal values.

F. *Hyperparameters*

The hyperparameters for Centralized DoubleDQN are tuned based on similar criteria as in Project 2. As shown in Table 2, the values of hyperparameters are very similar to those I used in Project 2. The major modification is an increase in the depth of the network to have three hidden layers, and the width of the first two hidden layers is doubled. While this increases the training time cost, I found that the model can converge faster. Another significant update is the change in the learning rate to be very small, allowing the weights of the network to be updated in small steps to avoid overlooking possible optimal cases. After decreasing the learning rate, the model converges faster. The minibatch size is also increased to 256 to ensure enough data can be learned and to have smooth updates.

TABLE II: Tuned values for hyperparameters

Hyperparameter	Tuned Value
maximum NUM_EPISODES	2000
TIME_STEPS	400
minibatch size BATCH_SIZE	256
γ	0.99
start ϵ	1.0
MIN_EPSILON	0.001
EPSILON_ DECAY_FACTOR	0.996
Network learning rate lr	0.00005
Network hidden layer 1 size	256
Network hidden layer 2 size	128
Network hidden layer 3 size	64
Network loss function	MSE-Mean Squared Error
Network optimizer	Adam
Soft Update Averaging Factor τ	0.001
Gradient Clipping Max Norm	20.0

IV. Results and Discussion

A. *Compare Centralized DDQN with Independent DDQN*

In this section, the performance of Centralized DDQN and Independent DDQN is compared for the layouts 'cramped room' and 'asymmetric advantages' to demonstrate that Centralized DDQN could achieve better performance than Independent DDQN.

As illustrated in Figures 1 and 2, for both layouts, Centralized DDQN converges faster than the independent approach. Moreover, for the 'asymmetric advantages' layout, the centralized method can even achieve a higher maximum number of soups delivered in 2000 episodes. In the case of 'asymmetric advantages,' the two agents are separated, and there are two pots compared to 'cramped room,' which only has one pot. The usage of the centralized approach helps the agent optimally use either of the pots to cook. Therefore, the number of soups after convergence is increased by 1.

After this comparison, I will focus on discussing the performance of Centralized DDQN for other layouts since it generally yields better performance.

B. *Training Result for 5 Layouts using CentralizedDDQN: success and failure*

As shown in Figure 3, the training for the layouts 'cramped room', 'asymmetric advantages', and 'coordination ring' successfully converges within 2000 episodes. However, for the 'forced coordination' layout, the agents show no signs of learning at all. In the case of the 'counter circuit o 1order' layout, the agents seem to start learning around episode 1200 to have the max number of soup to be 7, but then gradually collapse, stop to learn at all by episode 1600.

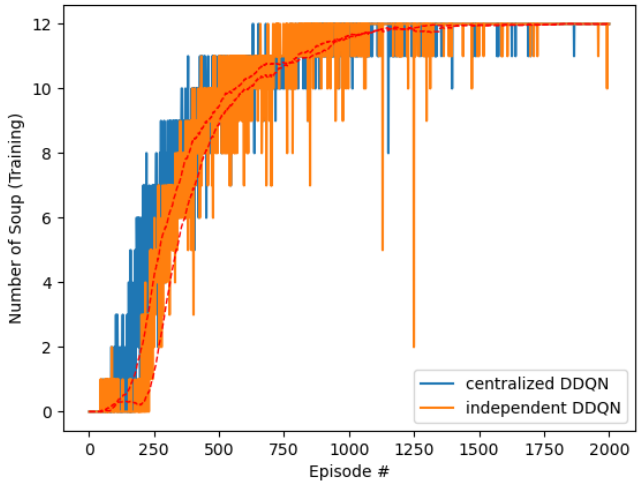

Fig. 1: Centralized vs. Independent-Train: cramped room

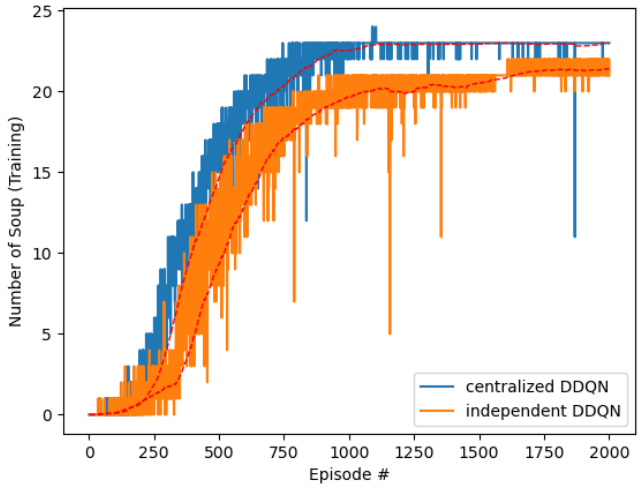

Fig. 2: Centralized vs. Independent-Train: asymmetric advantages

The outcomes of 'cramped room' and 'asymmetric advantages' are explained and discussed in the last subsection.

The good performance on 'coordination ring' demonstrates that the centralized approach can support multi-agents to learn cooperation to a certain extent.

However, for the layout 'forced coordination,' which absolutely requires coordination between agents in cooking every single soup, the approach used in this work cannot meet this high level of cooperation requirement. The design for handling coordination between agents in this centralized DDQN is not sufficient. This may be due to the fact that every reward is generated by two actions from two agents together. The approach in this work is not effectively learning this relationship between the reward and the joint actions. Possible improvements for this approach include modifying the Q-network learning to learn from both the reward and

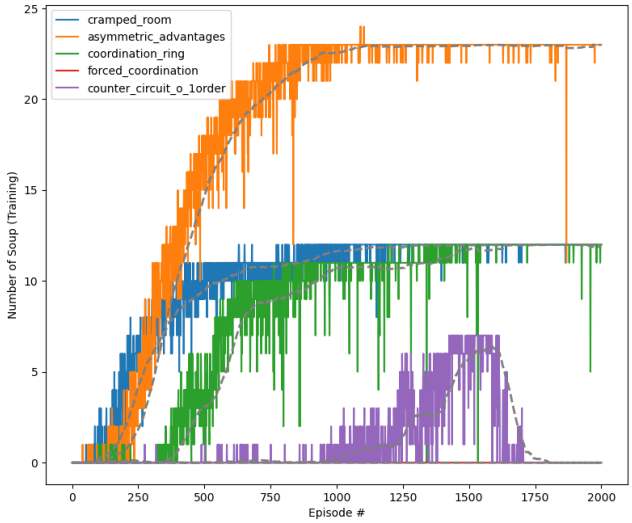

Fig. 3: Training Result for 5 Layouts

joint actions, instead of only learning the reward with separate actions. Additionally, the reward shaping could be modified to include some rewards reflecting the impact of the position between agents and the relative position between each agent and objects. Specifically for the 'forced coordination' layout, since one agent cannot access the pot and serving area at all, and another agent cannot access the dish and onion at all, studying the distance between agents with certain areas and objects will be essential for cooperation. For instance, if after checking the distance between an agent and the onion dispenser at every time step, we find out that this agent can never go near the onion. Then this agent accumulates no reward from this state information, making the network learn to avoid this agent attempting to reach the onion in the first place.

For the 'counter circuit o 1order' layout, the reward dropped from 7 to 0 at episode 1600. This could be due to the fact that this layout might require a lengthy training process, but my ϵ decays too aggressively for this layout. Therefore, the model doesn't get the chance to explore most parts of the system before ϵ drops to a very small value. Consequently, the agent becomes biased by suboptimal actions learned by the network. Slowing down the decay process might help with this layout in future work.

However, since all the layouts need to have the same approach and setup, I ran out of time to optimize the setup based on the discussion above and rerun the 5 layouts together. Nevertheless, I do believe these improvements could be helpful based on the discussions.

C. *Evaluation Result for 5 Layouts using CentralizedDDQN*

For the testing process of 5 layout in 100 episodes, as shown in Figure 4, the trained agents on first 3 layouts generalize

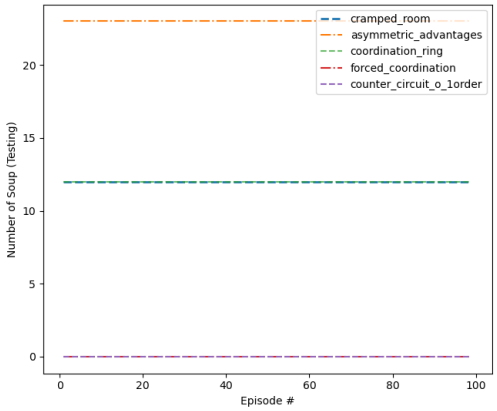

Fig. 4: Testing Result for 5 Layouts

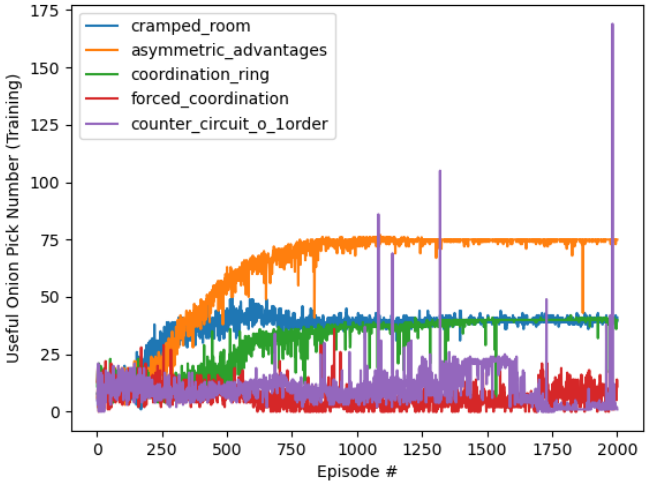

Fig. 5: Useful Onion Pickup Number - Training

pretty good. For the last 2 layouts, since the training failed, the number of soups during testing are 0s.

D. Two Other Metrices: Useful Onion Pickup Number and Useful Dish Pickup Number

As shown in Figures 5 and 6 for the two metrics 'Useful Onion Pickup Number' and 'Useful Dish Pickup Number' during training, the trend of these two metrics strongly correlates with the training reward data in Figure 3. After studying the resulting metrics corresponding to other events presented in reward shaping, it was found that these two metrics should be the events we want to reward with additional points at each time step. Due to my time limitation, I didn't get a chance to modify the reward shaping and reward these two events more, and rerun the experiments. However, this strong correlation evidence indicates that this improvement would benefit the training process and final achievable number of delivered soups.

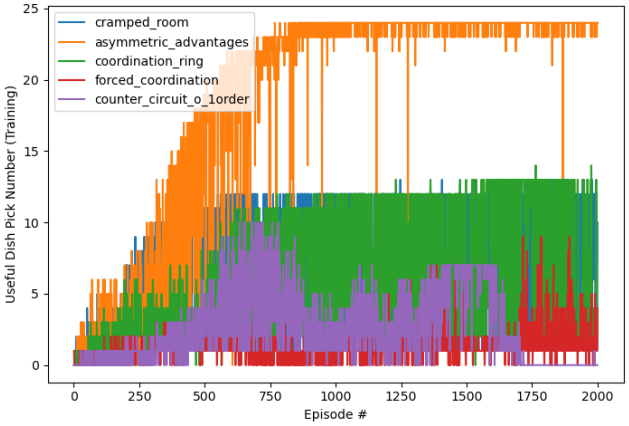

Fig. 6: Useful Dish Pickup Number - Training

V. Pitfalls and Future Work

The biggest pitfall I encountered is the reward swapping bug in this environment. When I first saw the description of this bug in Ed, I thought we needed to swap the states and actions, and the training and testing results were very unstable. Then I realized that we needed to swap the rewards. Another pitfall is the time limitation. The training process took a very long time and limited the flexibility to try more approach configurations and algorithms.

Based on the **Results and Discussion Section**, in the future, the algorithm could be improved by:

- modify reward shaping include some rewards reflecting the impact of the position between agents and the relative position between each agent and objects;
- modify Q-network learning to learn from the reward and the joint-actions;
- slow down the ϵ decay process;
- modify reward shaping to level up the granted reward for two events: Useful Onion Pickup and Useful Dish Pickup;
- try other algorithms (e.g., RIAL, DIAL, MDDPG, MPPO).

References

[1] https://github.com/HumanCompatibleAI/overcooked_ai/tree/master
[2] Hasselt et al. "Deep Reinforcement Learning with Double Q-learning," AAAI 2016.
[3] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning," NIPS Deep Learning Workshop 2013.
[4] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning," Nature volume 518, pages529–533 (2015).
[5] Jakob Foerster et al. "Learning to Communicate with Deep Multi-Agent Reinforcement Learning". In: Advances in Neural Information Processing Systems 29 (2016), pp. 2137–2145.
[6] Yaodong Yang et al. "An Overview of Multi-agent Reinforcement Learning from Game Theoretical Perspective"
[7] Jakob N. Foerster et al. "Counterfactual Multi-Agent Policy Gradients". In: CoRR abs/1705.08926 (2017). arXiv: 1705.08926. url: http://arxiv.org/abs/1705.08926.
[8] Ryan Lowe et al. "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments". In: CoRR abs/1706.02275 (2017). arXiv: 1706.02275. url: http://arxiv.org/abs/1706.02275.
[9] https://medium.com/@nerdjock/deep-learning-course-lesson-10-6-gradient-clipping-694dbb1cca4c: :text=Gradient