Multi-Agent Reinforcement Learning on Overcooked AI

Louis Wong
Georgia Tech
lwong64@gatech.edu

GitHub-Hash: 7ea338d65af896f7095b941d0c8ee6e06e5618cb

Abstract

This study explores the deployment of various reinforcement learning algorithms, including Deep Q-Network (DQN) and Value Decomposition Network (VDN), within the multi-agent 'Overcooked' game environment. By simulating agents tasked with cooperatively preparing and delivering onion soups across different kitchen layouts, we assess the impact of coordination and individual agent strategies on performance. Our findings reveal the critical role of hyperparameters and the structure of learning networks in achieving efficient decision-making and adaptability in dynamic settings. This investigation enhances understanding of multi-agent systems in complex environments and informs strategies for tuning algorithms for better stability and performance.

1. Environment and Task Description

In this project, agents are trained to prepare and deliver onion soups within the **Overcooked** game environment. The task spans five distinct kitchen layouts: cramped room, asymmetric advantages, coordination ring, forced coordination, and counter circuit. Each layout presents unique challenges, requiring both individual and cooperative strategies. The primary objective is to maximize soup deliveries within a 400-timestep episode, with each soup requiring 20 timesteps to cook. Delivering a completed soup garners a +20 reward. Inefficiencies such as cooking with insufficient ingredients, dropping a soup, or misplacing a delivery incur no penalties but result in lost time [1].

1.1. State Space

The Overcooked environment operates as a fully-observable Markov Decision Process (MDP), where both agents have access to complete state and observation spaces, represented identically as a 96-element vector for each agent. This vector comprises:

- **Player Features** (46 elements each): Includes orientation, object held, distances to nearest items (onion, dish, soup), and detailed pot-related attributes such as existence, state (empty, full, cooking, ready), contents, and proximity. Walls and other immediate player obstacles are also encoded.
- **Other Player Features** (46 elements): A mirrored set of the first player's features for the second player.
- **Inter-player Metrics** (4 elements): Distance and position relative to each other.

This encoding contain each agent's data includes not only their immediate surroundings and state but also a reflection of the other agent's situation, coordinated actions in the game environment. The layout specifics, such as the presence and state of cooking pots and walls, are tailored per scenario, enhancing the agents' decision-making processes based on the environment's current configuration, for more information refer to **Overcooked Documentation**.

1.2. Action Space

The discrete action space consists of six actions:

0. Move up
1. Move down
2. Move left
3. Move right
4. Stay
5. Interact (context-sensitive)

1.3. Objective

The challenge is to develop a single algorithmic approach that is effective across all layouts, aiming for a minimum average of seven delivered soups per episode. This approach must be robust enough to adapt to the various strategic demands of each layout, testing the agents' ability to optimize their movements and interactions within a constrained timeframe.

1.4. Multi-Agent Coordination

Figure 1. Various Overcooked layouts: "cramped room," "asymmetric advantages," "coordination ring," and "counter circuit."

Given the varying layout complexities, some of which may only be solvable through cooperative strategies, the approach must inherently support multi-agent collaboration. The analysis will also include evaluating multi-agent metrics to assess the effectiveness of the coordination strategies employed shown in Figure 1.

This environment provides a rigorous test of an AI's ability to generalize across different scenarios using a fixed set of strategies and parameters, emphasizing the importance of adaptability and efficient decision-making in dynamic settings.[2]

2. Initial Approach

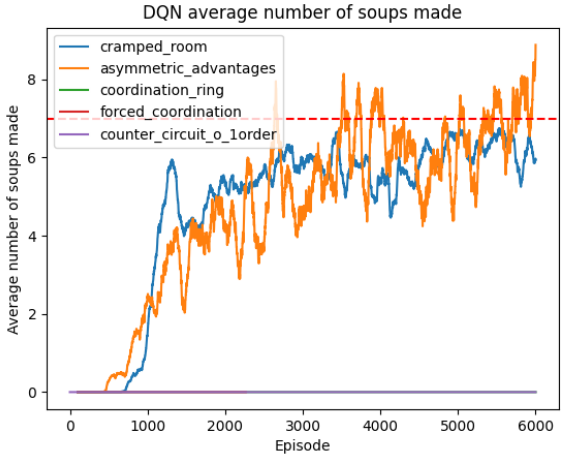

Figure 2. DQN average 100 episode soups made.

Initially, a simplified Deep Q-Network (DQN) from the predecessor project Lunar Lander was used. This model utilized a single neural network for both update and target calculations. For a game like Lunar Lander, it was sufficient to obtain a good benchmark shown in Figure 2. We then tested this model in a multi-agent reinforcement learning environment, where we instantiated two independent agents, each using the same DQN class. As a result, we had two independent DQNs, referred to as IDQNs. They achieved moderate success, with results above or close to the benchmark on "Cramp Room" and "Symmetrical Advantage" maps without any fine-tuning. However, the model fell short on the remaining three maps, due to its coordination setup.

3. Coordination

One of the primary reasons the remaining three maps were difficult for our IDQN agents to learn was the high amount of coordination required. The maps were designed in a single-path circular manner, which easily allowed agents to block each other's paths, making it inefficient to perform tasks. As reinforcement agents that act independently, where each agent only focuses on optimizing its own reward, they did not cooperate as a team. This was a tremendous challenge in this type of multi-agent reinforcement learning (MARL) environment. To succeed across all five maps, a structure that encouraged such collaboration and coordination with a common goal was necessary.

4. Value Decomposition Network (VDN)

One way to achieve collaboration was through the Value Decomposition Network (VDN), introduced by the Google DeepMind team in 2017. Instead of each agent learning and operating independently, VDN uses a single joint reward signal for centralized training, also known as Centralized Training with Decentralized Execution (CTDE). Conceptually, each agent maintains its own policy but shares one value function. The reward of the value function is the sum of the values from two agents:

$$V(s) = \sum_{i=1}^{n} Q_i(s, a_i) \tag{1}$$

$V(s)$ represents the total value at state s, and Q_i is the Q-value of agent i for its action a_i. There is no clear distinction between which value corresponds to which agent. Although it may seem counterintuitive at first, to assign credit properly, backpropagation and gradient flow are used. Ideally, any reward will flow back to influence both the policy and the features associated with it [4]

5. Double DQN (DDQN)

Double DQN is an advanced version of the simple DQN framework, which introduced two networks: the update network and the target network. Agents use the target network to estimate the Q-values but iteratively improve the update network. While doing this update, the target network is not updated because consistently updating the target network can lead to unstable learning, akin to constantly changing your predictor. Therefore, the target network is updated pe-

riodically and is a clone of the update network. This approach promotes the stability of learning:

$$Q_{\text{target}} = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^{-}) \quad (2)$$

Here, r represents the reward, γ is the discount factor, s' is the next state, a' is the next action, θ are the parameters of the update network, and θ^{-} are the parameters of the target network [3][5].

6. Experiment Results

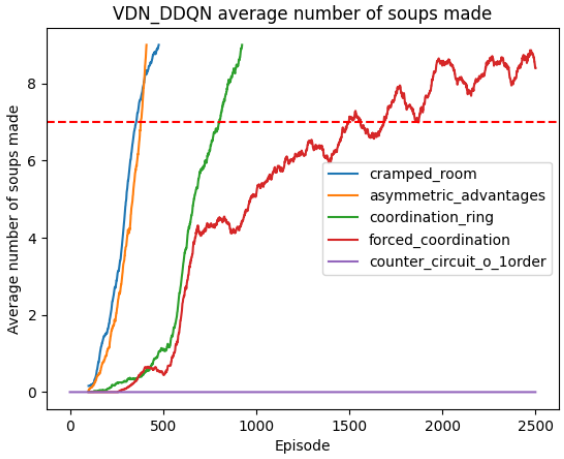

Figure 3. VDN-DDQN average 100 episode soups made.

Our Value Decomposition Network Double Deep Q-Network (VDN-DDQN) experiment approach successfully surpassed the benchmark across four layouts shown in Figure 3, namely: "cramped room," "asymmetric advantages," "coordination ring," except for "counter circuit." Generally, we found that multi-agent reinforcement learning poses a very challenging task, influenced by architecture, reward structure, and hyperparameters. The model is extremely sensitive to each component and part. Any deviation from the optimal values proves detrimental to overall learning. Through our experiences, we identified several crucial factors, including architectural design choices, reward structures, and hyperparameters such as learning rate, discount rate, exploration decay, replay memory buffer size, mini-batch size, and target network update frequency.

7. General Design Choices

For the DDQN network utilized in the VDN-DDQN, we chose a configuration of two fully connected hidden layers, each with 64 neurons with rectified linear unit (ReLU) activation. This design provides sufficient nonlinearity to facilitate effective learning without being so large as to extend training times unnecessarily. We employed Adam as our optimizer, known for its robustness and lesser sensitivity to learning rate variations compared to other optimizers, thereby offering greater flexibility in parameter adjustments.

The network's input size corresponds to the state observed by the agent, while the output size determines the actions available to the agent. The exploration rate was initially set to 1, with a decay rate applied to gradually decrease randomness through the Epsilon-Greedy algorithm. This setup addresses a regression problem aimed at predicting the Q-value function, with mean squared error as the chosen loss function.

In addition to the network architecture, we implemented reward shaping to assist the agent's learning process. Given the complexity of learning to make a soup through random steps alone, reward shaping provides intermediate rewards, thus promoting goal-oriented behavior. To ensure smooth performance of the predictor, we updated the target network only after every five episodes and replayed memories at every step, enhancing both learning stability and efficiency.

8. Replay Memory and Batch Size

Replay memory size is a critical choice in reinforcement learning. A small memory size risks losing a lot of valuable information collected by the agents. It is crucial to maintain a buffer size that retains many of the relevant states and rewards but also important to forget older memories. From our empirical experiments, we found that a maximum length of around 100,000 - 200,000 works effectively. According to *"Playing Atari with Deep Reinforcement Learning"* (2013) and its extended version *"Human-level control through deep reinforcement learning"* (2015), the standard approach involves maintaining a replay buffer that stores a fixed number of the most recent experiences. Once the buffer capacity is reached, older experiences are discarded to make room for new ones [3].

Another significant aspect of this experiment is the mini-batch size. In supervised learning, common batch sizes include 32, 64, and 128. However, in multi-agent reinforcement learning (MARL), larger mini-batch sizes enhance the learning experience. *"The Surprising Effectiveness of PPO in Multi-agent Games"* (2022) provided valuable insights, suggesting that generally, the larger the batch size, the better it is for the network to generalize and estimate the correct Q-value function, though it should be reduced for computational efficiency. We found an effective range of 256 to 512 for mini-batch size, which is close to half an episode, to a full episode.

9. Discount Rate

The discount factor (γ) plays a critical role in the learning process, especially in environments like Overcooked where agents receive rewards for sequential tasks such as adding ingredients to a soup or serving it. The maximum wait time for a soup to cook is approximately 20 steps, while adding an onion to the soup typically requires 5 to 10 steps. Consequently, we deduced that a lower discount rate, which focuses on a horizon of about 10 to 20 steps, would be more appropriate. A high discount rate might lead agents to overvalue experiences too distant in the future, potentially decreasing the quality of their action selection. We observed that discount rates ranging from 0.85 to 0.9 were effective.

10. Exploration vs. Exploitation

The exploration rate is a key factor influencing the success of agents. For simpler maps where agents can learn quickly, a lower exploration rate suffices. However, for more complex maps, a higher exploration rate is necessary. We fine-tuned the exploration decay rate based on empirical findings. Agents began learning to make soup after approximately 1,000 to 2,000 episodes, necessitating a decay rate that would still enable learning beyond these episodes. We found that decay rates of 0.99999 and 0.999995 per step effectively reduced the exploration rate to 0.01 and 0.13 after 1,000 episodes, respectively; however, the minimum exploration rate is kept at 0.1. It is maintained in this way to always leave room for the agent to explore in the later stages, which is critical to achieving stable results in benchmarking.

11. Gradient Clipping

Empirically, we start with a learning rate of 0.001 and later decrease it to 0.001 to increase the stability of the learning process. After a significant amount of learning, we've observed that the agent sometimes receives a very large reward, leading to fairly abrupt updates that cause the gradient to explode to very high values. Subsequently, the loss escalates, and the agent fails to recover. Therefore, it is crucial to implement a technique to prevent gradient explosion. Gradient clipping is one such technique, where we define a maximum threshold for each update to prevent overly large updates to the network. This is particularly important in Multi-Agent Reinforcement Learning (MARL), where we use a large mini batch size. The larger the batch size, the more prone it is to cause a large update, and a large update can significantly disrupt the gradient. Particularly, we noticed that the rewards from the environment are very sporadic. Often, the agent only receives a reward of zero, but suddenly it might receive a reward of 50 or even up to 100. During preliminary tests, we observed that with such large rewards, the maximum gradient tends to increase. We set the threshold initially at 100 and later reduced it to 25 to ensure safety. A threshold that is too low can slow down the learning process, but if it's too high, it can expose the gradient to the risk of explosion.

12. Frame Stacking

Our chosen hyperparameter successfully surpassed benchmarks for the layouts "cramped room," "asymmetric advantages," and "coordination ring." However, it faced challenges with layouts such as "forced coordination" and "counter circuit." The original Deep Q-Network (DQN), developed by DeepMind for playing Atari games, effectively utilized temporal information by incorporating a technique known as frame stacking. This approach provided the neural network with a stack of the last four frames as input, rather than a single frame. This enabled the DQN to perceive motion and temporal progression, which is crucial in games where understanding the sequence of past actions influences future decisions. For example, in "Breakout," this allowed the network to discern the ball's direction, offering a basic form of memory about recent game events [3].

In our approach, to avoid significantly increasing computational demands, we incorporated only the last observation along with the current one. Intuitively, a static picture doesn't reveal motion, but with just one previous frame, the motion becomes apparent. To our surprise, by adding just one additional observation and without changing any other parameter, we were able to exceed the benchmark in "forced coordination," achieving what was previously 0 soups served. This empirical finding underscores the significance of temporal information.

13. Afterthought

This experiment encountered numerous pitfalls, particularly how sensitive learning outcomes are to hyperparameter tuning. If even one hyperparameter is not tuned within an appropriate range, learning may not occur at all.

One significant technical challenge we encountered was related to the format of the state arrays provided by the environment. Initially, these arrays were in NumPy format, and for training with PyTorch, we needed to convert them into tensors. At first, this conversion process was performed each time the agent replayed and recalled from the memory buffer, leading to a very slow and repetitive process. To optimize this, we adjusted our approach so that each state array was converted to a tensor at the time of memorization. Specifically, each state and its corresponding reward value were converted into tensors during the memorize function

call, thus storing the data as tensors directly in the buffer. This adjustment significantly enhanced the efficiency of the replay process, especially when working with large mini-batches that randomize observations in the memory buffer. As a result, processing speed improved significantly.

Moreover, the stability of the gradient has a huge impacts learning quality. It is crucial to monitor the gradient closely during the learning process to ensure it contributes to incremental learning and remains stable without major fluctuations. Although we were able to identify approximate ranges for each hyperparameter, pinpointing the exact values across all conditions remains a challenge. Computation time and resources are significant constraints in this multi-agent reinforcement learning (MARL) environment. Given more computational resources and time, we would focus on searching for optimal values within these narrowed ranges using more systematic techniques, such as genetic algorithms. This would involve inheriting hyperparameter values from one map to the next, progressively refining them across all maps. Additionally, if project time constraints were less stringent, we could explore using architectures like recurrent neural networks (RNN) or long short-term memory networks (LSTM) instead of fully connected layers. These architectures could potentially yield superior results by better addressing the temporal aspects of the environment possibly without the necessity for extensive fine-tuning.

Appendix

Hyper-parameter	Value
Exploration Decay per Step	0.99999
Minimum Exploration Rate	0.1
Minibatch Size	512
Discount Rate	0.90
Max Norm	25.0
Memory Buffer Size	200,000
L2 Regularization	1×10^{-4}
Initial Exploration Rate	1
Update Target Every	2,000 steps

Table 1. VDN-DDQN Configuration Hyper-parameters

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint*, 2016. 1

[2] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10012–10022, 2021. 2

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. 3, 4

[4] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore Graepel. Value-decomposition networks for cooperative multi-agent learning. *arXiv preprint arXiv:1706.05296*, 2017. 2

[5] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2015. 3