MA: BeerGame

2100012521 崔绍洋

June 2024

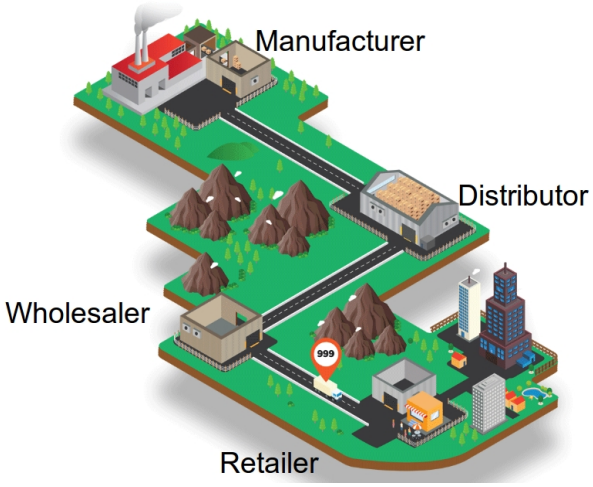


Figure 1: BEER GAME

1 Introduction

The Beer Game is a classic experimental scenario in multi-agent reinforcement learning. It abstracts the Retailer, wholesaler, distributor, and manufacturer into independent agents in a product supply chain and simplifies their functions to "place orders" and "ship and receive goods." In the interaction process, the goal is to fulfill customer orders as much as possible and maximize each agent's profit.

In our game scenario, we use the default reward settings in the demo[4]. Since the default settings do not offer positive rewards, the optimal profit obtained by the model should be equal to or close to zero.

- Holding cost for each agent: [-2, -2, -2, -2]
- Shortage cost for each agent: [-2, 0, 0, 0]

In our experiment, we select the Retailer as the primary agent and use Deep Q-learning (DQN)[2] for training and policy execution, while all other agents adopt the base-stock policy. Additionally, we explored the impact of different network structures, parameter combinations, and training strategies on the final experimental results. Furthermore, we aim to investigate whether DQN can still effectively solve the problem when the number of RL decision-making agents changes.

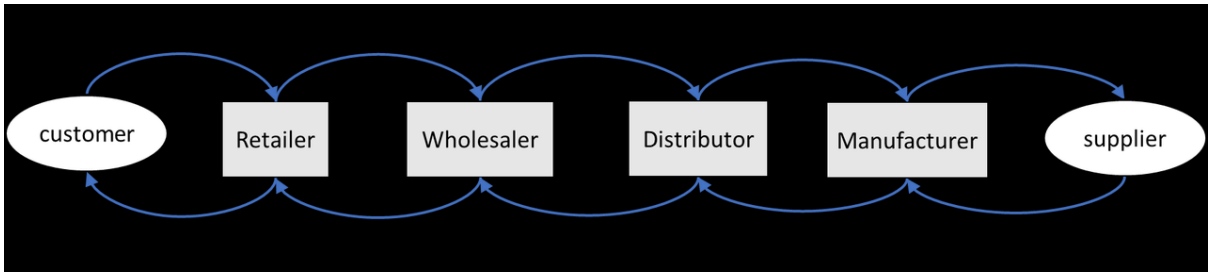


Figure 2: Chain of Beer Game

2 Method

Deep Q-learning (DQN) is the pioneering work in reinforcement learning based on Q-learning:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(R_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)) \tag{1}$$

In scenarios where the state and action spaces are too large, a deep neural network Q_θ is used to replace the Q-table for better learning and training. In our implementation, we use Double DQN[1], which involves training two different networks synchronously: the behavior network Q and the target network $\hat{Q}$:

$$Q_\theta(s_t, a_t) = Q_\theta(s_t, a_t) + \alpha(R_t + \gamma \max_{a'} \hat{Q}_{\theta^-}(s_{t+1}, a') - Q_\theta(s_t, a_t)) \tag{2}$$

where Q_θ and $\hat{Q}_{\theta^-}$ represent the behavior network and target network, respectively. During training, we sample data and use $r + \gamma \max_{a'} \hat{Q}_{\theta^-}(s_{t+1}, a')$ as the label and train the network using the MSE loss function. To ensure training stability, we update the weights of the target network every C time step: $\theta^- \leftarrow \theta$. We also use prioritized experience replay to ensure training stability and efficiency. The pseudo-code for the Double DQN algorithm is as follows:

Algorithm 1 Double DQN Algorithm

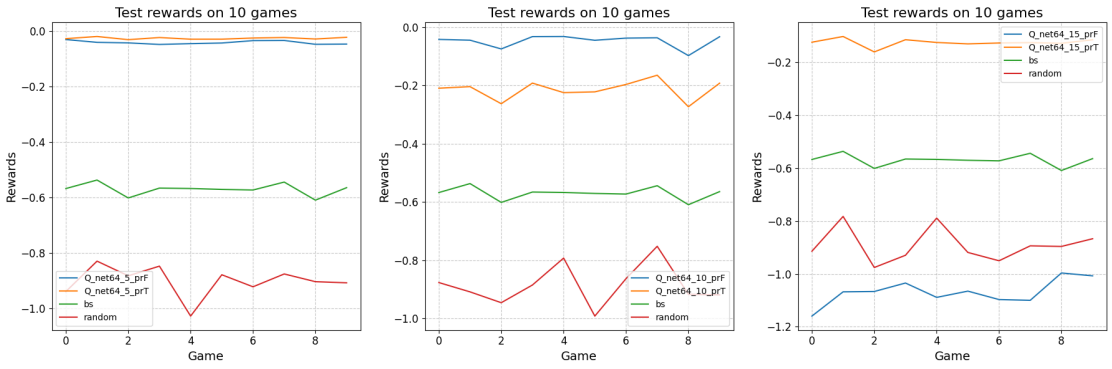
Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode = 1 to M **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for $t = 1$ to T **do**
        With probability ϵ select a random action a_t
        Otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
        Execute action a_t in the environment and observe reward r_t and next state s_{t+1}
        Set $\phi_{t+1} = \phi(s_{t+1})$, $p_i = \max_j(p_j \in \mathcal{D})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1}, p_i)$ in replay memory $\mathcal{D}$
        for $j = 1$ to $k = |\text{minibatch}|$ **do**
            Sample transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}, j \sim P(j) = \frac{p_j}{\sum_i p_i}$
            Set $y_j = r_j + \gamma \hat{Q}(\phi_{j+1}, \arg\max_a Q(\phi_{j+1}, a; \theta); \theta^-)$
            Perform a gradient descent step on $\delta_j = (y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
            Update the priority in buffer: $p_j \leftarrow \sqrt{\delta_j}$
        end for
        Every C steps, update target network: $\theta^- = \theta$
    end for
end for

We use the base-stock policy for other agents, maintaining inventory within a specific range. Each player sets a basic stock level for each product, typically based on historical demand data, forecasted demand, and supply chain lead time. When the inventory level drops to a certain threshold (reorder point), a replenishment order is triggered to return the inventory to the basic stock level.

3 Experiments

There are 50 tasks in total. We use the first 40 for training and the last 10 for testing to evaluate the effectiveness of our model. For the Q-Net model, we used a simple 3-layer MLP.

W selected $|\text{minibatch}| = k \in 5, 10, 15$ to train a Q-Net model for Retailer with other players using a base-stock strategy.



(a) k=5 (b) k=10 (c) k=15

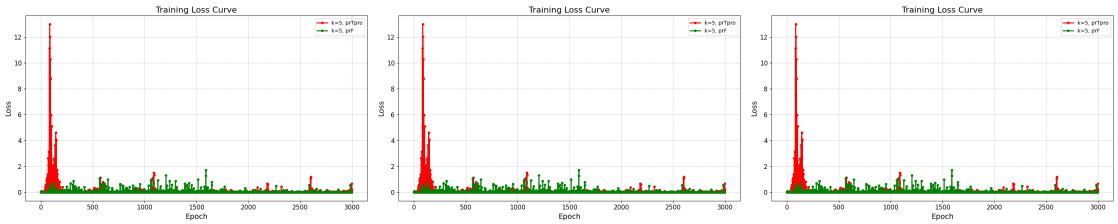
Figure 3: Loss curves with/without priority replay.

From the changes in loss with or without priority replay[3] shown in Figure 4, we can observe that priority replay causes severe fluctuations at the early stages of training compared to uniform sampling. The fluctuations may be due to the limited data in the buffer at the beginning of training, leading to repeated learning of certain parts of data and causing the model to fall into the local minimum. Although priority replay can make the loss convergence relatively stable in later stages, in the current problem context, it does not effectively enhance the stability of the training.

Then, we validated 10 test games and compared the average reward the Retailer got utilizing trained Q-Net with random and base-stock strategies(Table 1). In our case, $|\text{minibatch}| = 5$ and uniformly sample from buffer yields the highest rewards and lowest average loss simultaneously.

model/strategy	Reward (avg)	Average train loss
Base-stock	-0.5699	——
Random	-0.8630	——
DQN (k5)	**-0.0411**	**0.0113**
DQN (k10)	-0.0474	0.5723
DQN (k15)	-1.0683	5.2750
DQN (k5) + priority replay	-0.0447	0.0334
DQN (k10) + priority replay	-0.2284	2.6961
DQN (k15) + priority replay	-0.1352	8.5021

Table 1: Average Test Reward of Retailer using different strategies

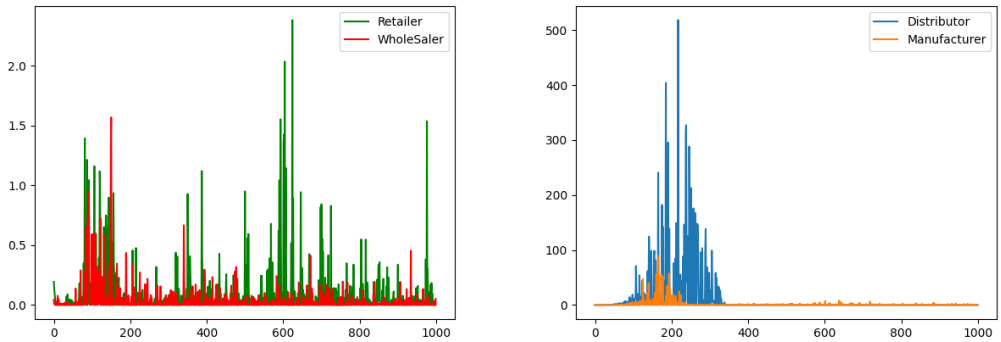


(a) k=5 (b) k=10 (c) k=15

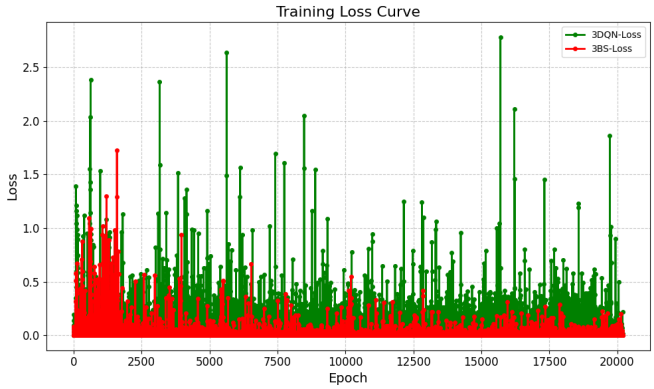
Figure 4: Loss curves with/without priority replay.

4 Discussion

Further, we aim to explore whether DQN can be applied to multi-agent reinforcement learning tasks. Therefore, all four agents involved in the Beer Game were trained simultaneously using the DQN algorithm. As shown in Figure 5, we observed that, overall, the roles at the back end of the supply chain (Figure 5b) exhibit greater training difficulty and higher loss volatility. Additionally, Figure 5c presents the training loss of the Retailer's Q-Net when faced with three DQN and three BaseStock strategy players. We found that as the number of training models and involved agents increased, the model's convergence deteriorated.



(a) Retailer&Wholesaler (b) Distributor&Manufacturer

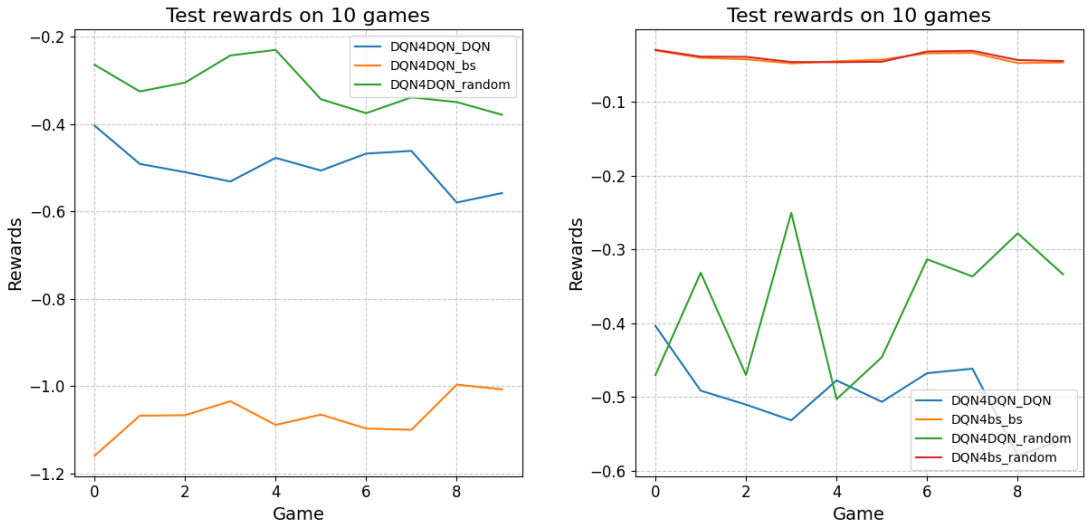


(c) 3DQN vs. 3BS

Figure 5: The training loss of all four Q-Nets. (a) Loss of Retailer and Wholesaler (b) Loss of Distributor and Manufacturer. Since there is an order of magnitude difference in loss between the first two and the latter two, we chose to display them in two separate figures.

We compared each player's reward results when all agents used DQN versus when each agent used DQN individually (as shown in the figure).

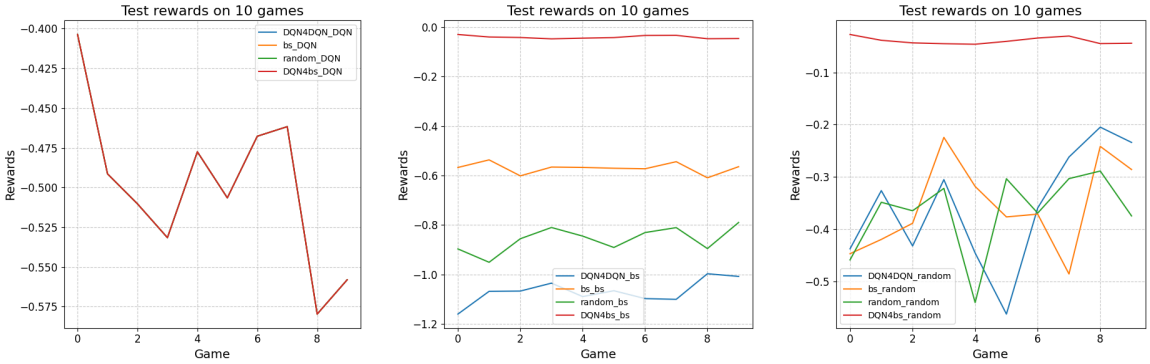
Additionally, we analyzed the performance of the learned Q_{retailer} (Figure 6). The results show that the Q_{retailer} trained with other players using DQN strategies performed poorly. Its model performed worse in the test set (other players also used the trained DQN for testing) than when facing random strategies. It indicates that in multi-agent games, simultaneously training multiple agents with deep reinforcement learning poses significant challenges, potentially leading to suboptimal results. Therefore, we may need to explore more powerful models, design more scientific and reasonable algorithms, and continuously improve the training process.



(a) (b)

Figure 6: Average test rewards for Retailer under different scenarios: (a) DQN trained with three DQN competitors, tested against three full DQN, BaseStock, or random strategy competitors; (b) Retailer-DQN trained with DQN and BaseStock competitors, tested against training competitors, with its performance against random strategies as a reference.

Furthermore, we encountered perplexing results during the experiments. Figure **??** shows the average rewards of the Retailer using all four strategies (QNet trained based on DQN and BaseStock, BaseStock, random) when facing all DQN, BaseStock, and random strategies from other players. Interestingly, when other players used the trained QNet strategy, we found that the test task results were identical regardless of the Retailer's chosen strategy (Figure 7a). The underlying reason for this remains unclear and requires further research and verification.



(a) DQN (b) Base-Stock (c) Random

Figure 7: Average rewards for Retailer using four strategies (QNet trained based on DQN and BaseStock, BaseStock, random) when facing all (a) DQN, (b) BaseStock, and (c) random strategies from other players.

We also analyzed the order data from four simultaneously trained DQN models over 10 test games (Figure **??**). The wholesaler and manufacturer rarely place orders, which aligns with our expectations. According to our reward settings, neither need to face shortage penalties, so it is reasonable for them to avoid placing orders to minimize holding costs. On the other hand. The retailer placed orders below the customer demand level, which corroborates the reward results we obtained previously. However, the distributor's orders are abnormally high, which we believe is due to the instability of the model training, leading to convergence to a local optimum.

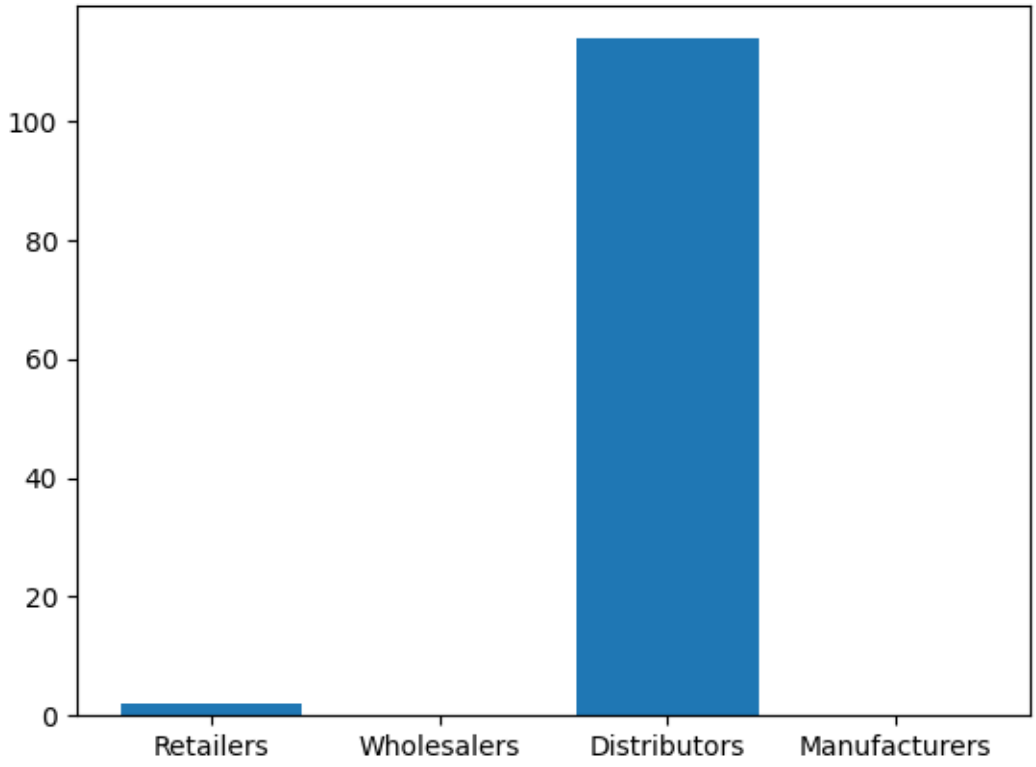


Figure 8: Average orders on 10 test games.

5 Configuration and Codes

Some of our configurations are listed in Table 2.

configs	values
γ	0.9
MLP-hidden layers	3
MLP-hidden size	64
optimizer	Adam
lr	0.001

Table 2: Configurations

We offered our codes for DQN on BeerGame demo in Codes.zip. To check it, here are some guidance.

Run: python new_env.py to train or test the Q-Net model, here are some arguments that might helps:

arguments	value	–help
–mode	train|test	train or test model
–model	model name	the model to be tested
–k	int	size of minibatch
–priority	bool	if use priority replay
–competitors	DQN|random|bs	The strategy of other players except Retailer.

References

[1] H. V. Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI Conference on Artificial Intelligence*, 2015.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Kirkeby Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[3] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[4] YanSong97. Repository title. https://github.com/YanSong97/BeerGame$_demo$, 2019.