# Online Supplements for A Deep Q-Network for the Beer Game: Reinforcement Learning for Inventory Optimization

## Appendix A: $\epsilon$-Greedy Algorithm

The $\epsilon$-greedy algorithm chooses a random actions with probability $\epsilon$, and with probability $1-\epsilon_t$ chooses the action greedily by evaluating a given function. In the context of the Q-Learning algorithm, with probability $\epsilon_t$ in time $t$, the algorithm chooses an action randomly, and with probability $1-\epsilon_t$, it chooses the action with the highest cumulative action value, i.e., $a_{t+1} = \text{argmax}_a Q(s_{t+1}, a)$. The random selection of actions, called exploration, allows the agent to explore the solution space and gives an optimality guarantee to the Q-Learning algorithm if $\epsilon_t \to 0$ when $t \to \infty$ (Sutton and Barto 1998).

## Appendix B: Sterman Formula Parameters

The computational experiments that use `Strm` agents calculate the order quantity using $q_t^i = \max\{0, AO_{t+1}^{i-1} + \alpha^i(IL_t^i - a^i) + \beta^i(OO_t^i - b^i)\}$, adapted from Sterman (1989), where $\alpha^i$, $a^i$, $\beta^i$, and $b^i$ are the parameters corresponding to the inventory level and on-order quantity. The idea is that the agent sets the order quantity equal to the demand forecast plus two terms that represent adjustments that the agent makes based on the deviations between its current inventory level (resp., on-order quantity) and a target value $a^i$ (resp., $b^i$). We set $a^i = \mu_d$, where $\mu_d$ is the average demand; $b^i = \mu_d(l_i^{fi} + l_i^{tr})$; $\alpha^i = -0.5$; and $\beta^i = -0.2$ for all agents $i = 1, 2, 3, 4$. The negative $\alpha$ and $\beta$ mean that the player over-orders when the inventory level or on-order quantity fall below the target value $a_i$ or $b_i$.

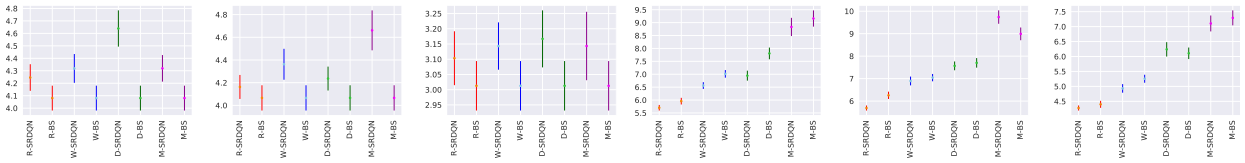
## Appendix C: Determining the Value of $m$ in the State Definition

As noted above, the DNN maintains information from the most recent $m$ periods in order to keep the size of the state variable fixed and to address the issue with the delayed observation of the reward. In order to select an appropriate value for $m$, one has to consider the value of the lead times throughout the game. First, when agent $i$ takes action $a_t^i$ at time $t$, it does not observe its effect until at least $l_i^{tr} + l_i^{in}$ periods later, when the order may be received. Moreover, agent $i+1$ may not have enough stock to satisfy the order immediately, in which case the shipment is delayed and in the worst case agent $i$ might not observe the corresponding reward $r_t^i$ until $\sum_{j=i}^4 (l_j^{tr} + l_j^{in})$ periods later. However, one needs the reward $r_t^i$ to evaluate the action $a_t^i$ taken. Thus, ideally $m$ should be chosen at least as large as $\sum_{j=1}^4 (l_j^{tr} + l_j^{in})$. On the other hand, this value can be large and selecting a large value for $m$ results in a large input size for the DNN, which increases the training time. Therefore, selecting $m$ is a trade-off between accuracy and computation time, and $m$ should be selected according to the required level of accuracy and the available computation power. In our numerical experiment, $\sum_{j=1}^4 (l_j^{tr} + l_j^{in}) = 15$ or 16, and we test $m \in \{5, 10\}$.

## Appendix D: Why Can't Standard DQN Solve the Beer Game?

One naive approach to extend the DQN algorithm to solve the beer game is to use multiple DQNs, one for each agent. However, using DQN as the decision maker for each agent results in a competitive game in which each DQN agent plays independently to minimize its own cost. For example, consider a beer game in which

**Figure 1** **Confidence intervals for the real-world dataset I.**



(a) Category 6, BS (b) Category 13, BS (c) Category 22, BS (d) Category 6, Strm (e) Category 13, Strm (f) Category 22, Strm

**Figure 2** **Confidence intervals for the real-world dataset II.**

(a) Category 5, BS (b) Category 34, BS (c) Category 46, BS (d) Category 5, Strm (e) Category 34, Strm (f) Category 46, Strm

players 2, 3, and 4 each have a stand-alone, well-trained DQN and the retailer (stage 1) uses a base-stock policy to make decisions. If the holding costs are positive for all players and the stockout cost is positive only for the retailer (as is common in the beer game), then the DQN at agents 2, 3, and 4 will return an optimal order quantity of 0 in every period, since on-hand inventory hurts the objective function for these players, but stockouts do not. This is a byproduct of the independent DQN agents minimizing their own costs without considering the total cost, which is obviously not an optimal solution for the system as a whole.

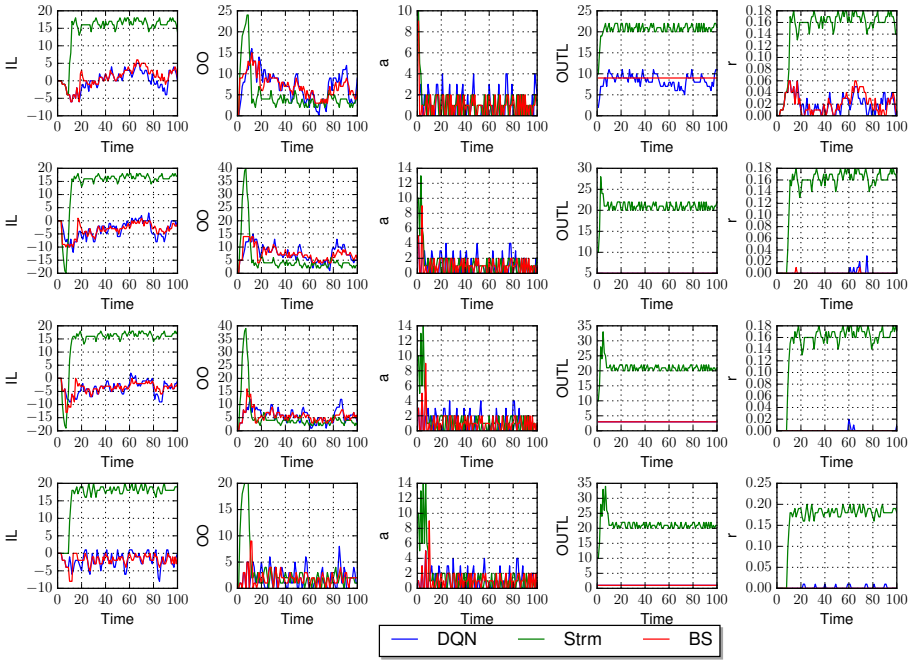
## Appendix E: Experience Replay

The DNN algorithm requires a mini-batch of input and the corresponding set of output values to learn the Q-values. Since SRDQN is an off-policy algorithm, we can use the new state $s_{t+1}$, the current state $s_t$, the action $a_t$ taken, and the observed reward $r_t$, in each period $t$. This information can provide the required set of input and output for the DNN; however, the resulting sequence of observations from the RL results in a non-stationary dataset with a strong correlation among consecutive records. This makes the DNN and, as a result, the RL prone to over-fitting the previously observed records and may even result in a diverging approximator (Sutton and Barto 1998). To avoid this problem, we use *experience replay* $E^i$ (Lin 1992) for agent $i$: We add observation $e_t^i = (s_t^i, a_t^i, r_t^i, s_{t+1}^i)$ iteratively into $E_i$, and in each training step, we take a random mini-batch of the experience to break correlations among the training data, thereby reducing the variance of the output.

## Appendix F: Confidence Interval for Real-World Dataset I and II

This appendix presents the 90% confidence intervals for all cases of the real-world dataset I and II. As it is shown in Figures 1 and 2, in most of `Strm` co-players, `SRDQN` outperform `BS`, and with the `BS` co-players, in 12 cases `BS-BS` obtains statistically smaller cost and in 12 cases `SRDQN` and `BS` obtains statistically equal costs.

**Figure 3** $IL_t$, $OO_t$, $a_t$, and $r_t$ of all agents when SRDQN retailer plays with three base-stock co-players (`BS-SRDQN`).




## Appendix G: Extended Numerical Results

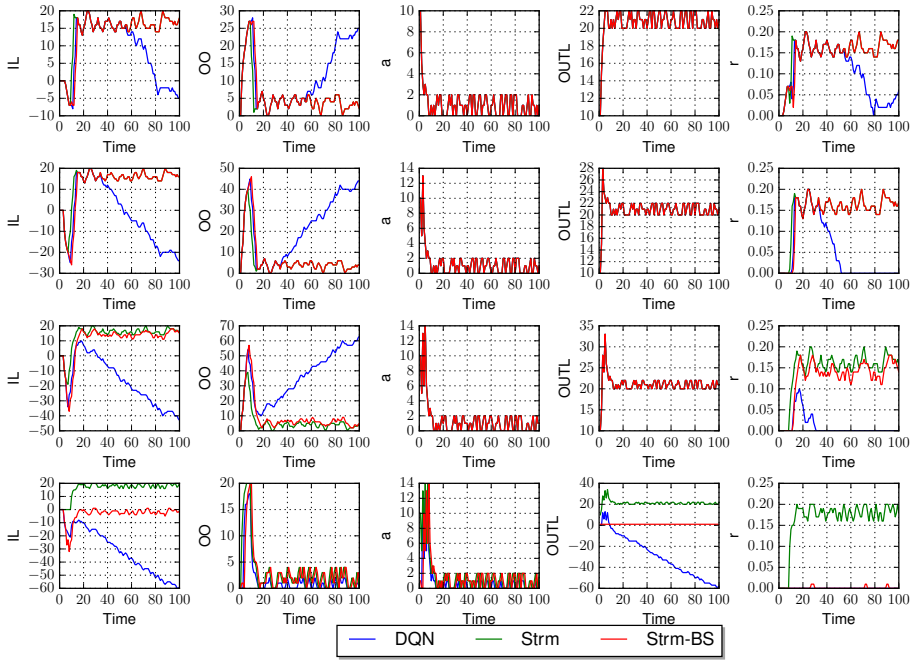
This appendix shows additional results on the details of play of each agent. Figure 3 provides the details of $IL$, $OO$, $a$, $r$, and OUTL for each agent when the SRDQN retailer plays with co-players who use the base-stock levels obtained by the Clark–Scarf algorithm. Clearly, `BS-SRDQN` attains a similar IL, OO, action, and reward to those of `BS-BS`. Figure 4 provides analogous results for the case in which the SRDQN manufacturer plays with three `Strm` agents. The SRDQN agent learns that the shortage costs of the non-retailer agents are zero and exploits that fact to reduce the total cost. In each of the figures, the top set of charts provides the results of the retailer, followed by the warehouse, distributor, and manufacturer.

## Appendix H: The Effect of $\beta$ on the Performance of Each Agent

Figure 5 plots the training trajectories for SRDQN agents playing with three agents that use the base-stock levels obtained by the Clark–Scarf algorithm, using various values of $C$, $m$, and $\beta$. In each sub-figure, the blue line denotes the result when all players use a base-stock policy while the remaining curves each represent the agent using SRDQN with different values of $C$, $\beta$, and $m$, trained for 60000 episodes with a learning rate of 0.00025.

As shown in Figure 5a, when the SRDQN plays the retailer, $\beta_1 \in \{20, 40\}$ works well, and $\beta_1 = 40$ provides the best results. As we move upstream in the supply chain (warehouse, then distributor, then manufacturer), smaller $\beta$ values become more effective (see Figures 5b–5d). Recall that the retailer bears the largest share of the optimal expected cost per period, and as a result it needs a larger $\beta$ than the other agents. Not surprisingly, larger $m$ values attain better costs since the SRDQN has more knowledge of the environment.

**Figure 4** $IL_t$, $OO_t$, $a_t$, and $r_t$ of all agents when SRDQN manufacturer plays with three Sterman co-players (`Strm-SRDQN`).




Finally, larger $C$ works better and provides a stable SRDQN model. However, there are some combinations for which smaller $C$ and $m$ also work well, e.g., see Figure 5d, trajectory 5000-20-5.

## Appendix I: Extended Results on Transfer Learning

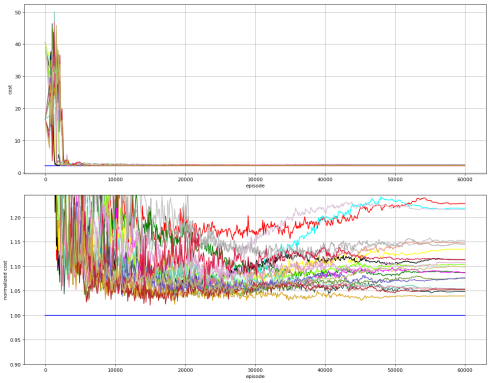
### I.1. Transfer Knowledge Between Agents

In this section, we present the results of the transfer learning method when the trained agent $i \in \{1,2,3,4\}$ transfers its first $k \in \{1,2,3\}$ layer(s) into co-player agent $j \in \{1,2,3,4\}$, $j \neq i$. For each target-agent $j$, Figure 6 shows the results for the best source-agent $i$ and the number of shared layers $k$, out of the 9 possible choices for $i$ and $k$. In the sub-figure captions, the notation $j$-$i$-$k$ indicates that source-agent $i$ shares weights of the first $k$ layers with target-agent $j$, so that those $k$ layers remain non-trainable.

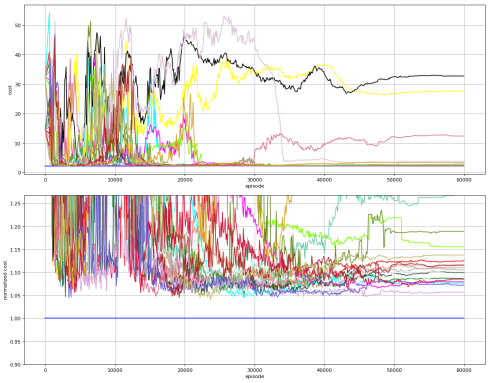
Except for agent 2, all agents obtain costs that are very close to those of the base-stock policy, with a 6.06% gap, on average. (In Section **??**, the average gap is 2.31%.) However, none of the agents was a good source for agent 2. It seems that the acquired knowledge of other agents is not enough to get a good solution for this agent, or the feature space that agent 2 explores is different from other agents, so that it cannot get a solution whose cost is close to the `BS-BS` cost.

In order to get more insight, consider Figure **??**, which presents the best results obtained through hyper-parameter tuning for each agent. In that figure, all agents start the training with a large cost value, and after 25000 fluctuating iterations, each converges to a stable solution. In contrast, in Figure 6, each agent starts from a relatively small cost value, and after a few thousand training episodes converges to the final solution. Moreover, for agent 3, the final cost of the transfer learning solution is smaller than that obtained

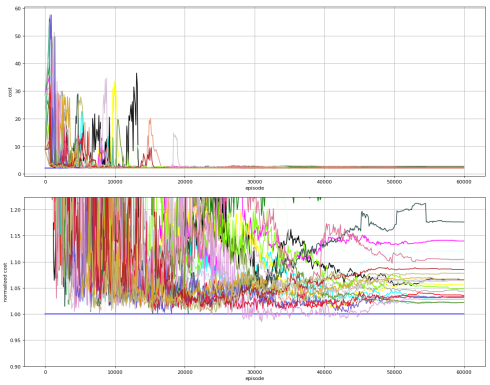
**Figure 5** **Total cost (upper figure) and normalized cost (lower figure) for `BS-SRDQN`.**



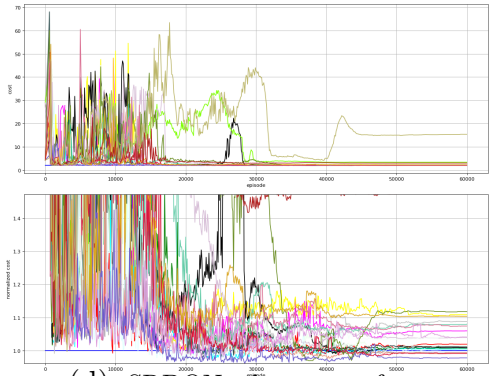
(a) SRDQN plays retailer



(b) SRDQN plays warehouse



(c) SRDQN plays distributor



(d) SRDQN plays manufacturer

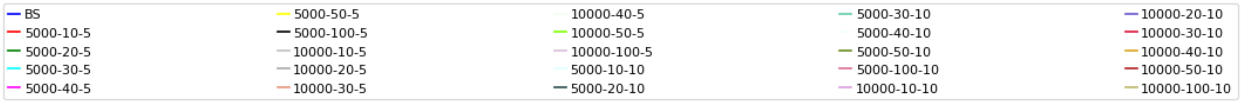
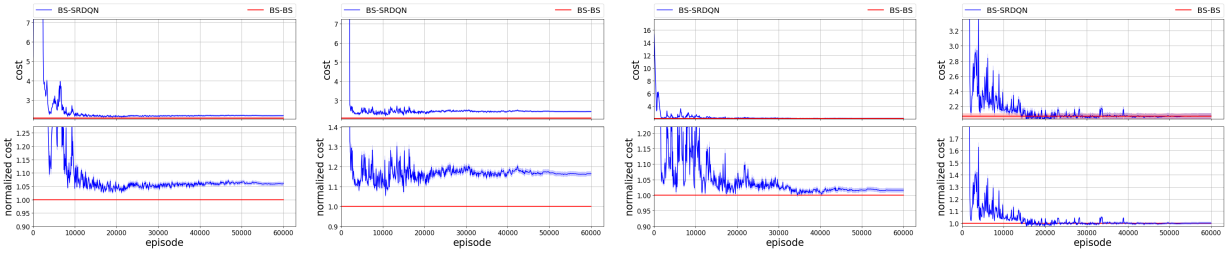



**Figure 6** **Results of transfer learning between agents with the same cost coefficients and action space.**

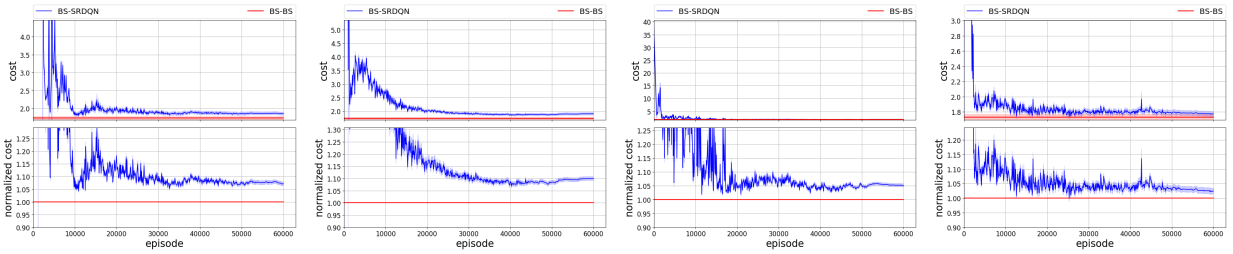



(a) Case 1-4-1 (b) Case 2-4-1 (c) Case 3-1-1 (d) Case 4-2-1

by training the network from scratch. And, the transfer learning method used one order of magnitude less CPU time than the approach in Section **??** to obtain very close results.

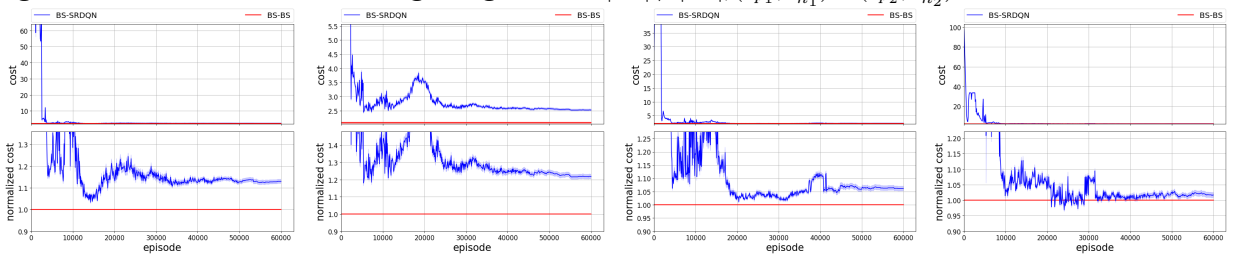
We also observe that agent $j$ can obtain good results when $k = 1$ and $i$ is either $j-1$ or $j+1$. This shows that the learned weights of the first SRDQN network layer are general enough to transfer knowledge to the other agents, and also that the learned knowledge of neighboring agents is similar. Also, for any agent $j$,

**Figure 7** **Fesults of transfer learning between agents with different cost coefficients and same action space.**




(a) Case 1-4-1 (b) Case 2-3-3 (c) Case 3-1-1 (d) Case 4-4-2

**Figure 8** **Results of transfer learning for agents with** $|\mathcal{A}_1| \neq |\mathcal{A}_2|, (c^j_{p_1}, c^j_{h_1}) = (c^j_{p_2}, c^j_{h_2})$.




(a) Case 1-3-1 (b) Case 2-3-2 (c) Case 3-4-2 (d) Case 4-2-1

agent $i = 1$ provides similar results to that of agent $i = j - 1$ or $i = j + 1$ does, and in some cases it provides slightly smaller costs, which shows that agent 1 captures general feature values better than the others.

### I.2. Transfer Knowledge for Different Cost Coefficients

Figure 7 shows the best results achieved for all agents, when agent $j$ has different cost coefficients, $(c_{p_2}, c_{h_2}) \neq (c_{p_1}, c_{h_1})$. We test target agents $j \in \{1, 2, 3, 4\}$, such that the holding and shortage costs are (5,1), (5,0), (5,0), and (5,0) for agents 1 to 4, respectively. In all of these tests, the source and target agents have the same action spaces. All agents attain cost values close to the `BS-BS` cost; in fact, the overall average cost is 6.16% higher than the `BS-BS` cost.

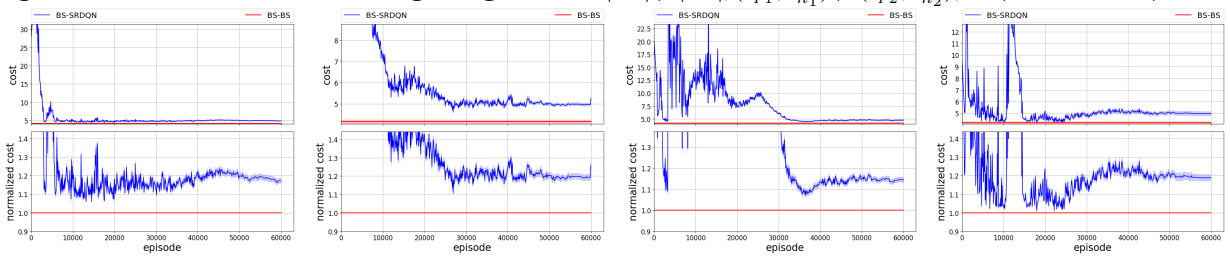
In addition, similar to the results of Section I.1, base agent $i = 1$ provides good results for all target agents. We also performed the same tests with shortage and holding costs (10,1), (1,0), (1,0), and (1,0) for agents 1 to 4, respectively, and obtained very similar results.

### I.3. Transfer Knowledge for Different Size of Action Space

Increasing the size of the action space should increase the accuracy of the $d + x$ approach. However, it makes the training process harder. It can be effective to train an agent with a small action space and then transfer the knowledge to an agent with a larger action space. To test this, we test target-agent $j \in \{1, 2, 3, 4\}$ with action space $\{-5, \ldots, 5\}$, assuming that the source and target agents have the same cost coefficients.

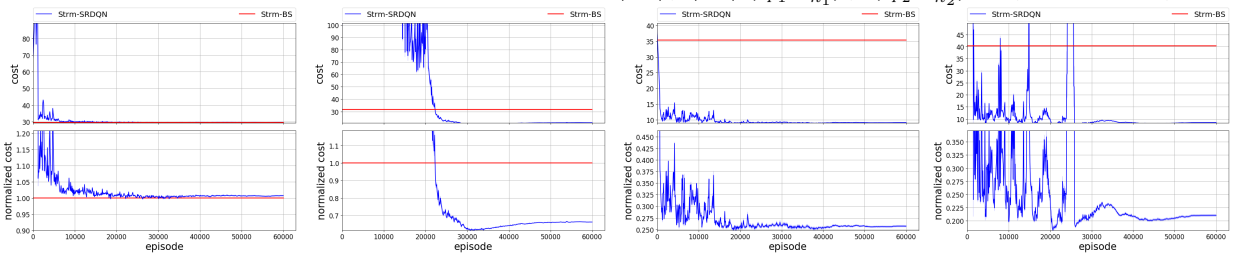
Figure 8 shows the best results achieved for all agents. All agents attained costs that are close to the `BS-BS` cost, with an average gap of approximately 10.66%.

**Figure 9** **Results of transfer learning for agents with $|\mathcal{A}_1| \neq |\mathcal{A}_2|, (c^j_{p_1}, c^j_{h_1}) \neq (c^j_{p_2}, c^j_{h_2}), D_1 \neq D_2$, and $\pi_1 \neq \pi_2$.**




(a) Case 1-3-1 (b) Case 2-3-3 (c) Case 3-2-1 (d) Case 4-3-2

**Figure 10** **Results of transfer learning for agents with $|\mathcal{A}_1| \neq |\mathcal{A}_2|, (c^j_{p_1}, c^j_{h_1}) \neq (c^j_{p_2}, c^j_{h_2}), D_1 \neq D_2$, and $\pi_1 \neq \pi_2$.**




(a) Case 1-1-1 (b) Case 2-1-3 (c) Case 3-1-1 (d) Case 4-1-1

### I.4. Transfer Knowledge for Different Action Space, Cost Coefficients, and Demand Distribution

This case includes all difficulties of the cases in Sections I.1, I.2, I.3, and **??**, in addition to the demand distributions being different. So, the range of demand, $IL$, $OO$, $AS$, and $AO$ that each agent observes is different than those of the base agent. Therefore, this is a hard case to train, and the average optimality gap is 17.41%; however, as Figure 9 depicts, the cost values decrease quickly and the training noise is quite small.

### I.5. Transfer Knowledge for Different Action Space, Cost Coefficients, Demand Distribution, and $\pi_2$

Figures 10 and 11 show the results of the most complex transfer learning cases that we tested. Although the SRDQN plays with non-rational (Sterman ) co-players and the observations in each state might be quite noisy, there are relatively small fluctuations in the training, and for all agents after around 40,000 iterations they converge.

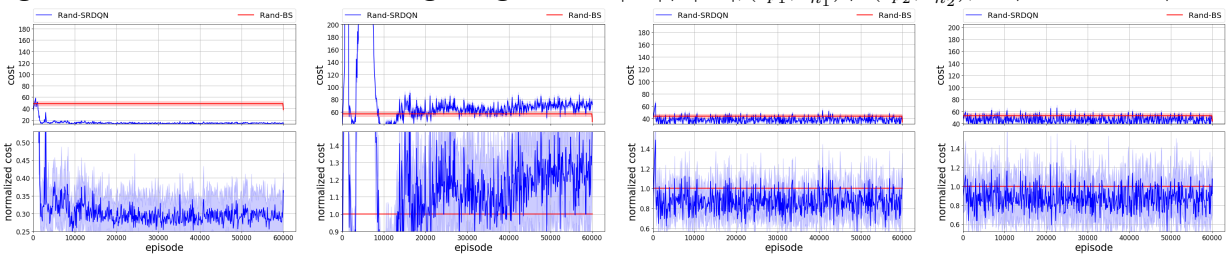
## Appendix J: Pseudocode of the Beer Game Simulator

The SRDQN algorithm needs to interact with the environment, so that for each state and action, the environment should return the reward and the next state. We simulate the beer game environment using Algorithm 1. In addition to the notation defined earlier, the algorithm also uses the following notation:

$d^t$: The demand of the customer in period $t$.

$OS^t_i$: Outbound shipment from agent $i$ (to agent $i-1$) in period $t$.

**Figure 11** **Results of transfer learning for agents with $|\mathcal{A}_1| \neq |\mathcal{A}_2|, (c^j_{p_1}, c^j_{h_1}) \neq (c^j_{p_2}, c^j_{h_2}), D_1 \neq D_2$, and $\pi_1 \neq \pi_2$.**




(a) Case 1-2-1 (b) Case 2-1-2 (c) Case 3-3-3 (d) Case 4-1-1

**Algorithm 1** Beer Game Simulator Pseudocode.

**procedure** PLAYGAME
  Set $T$ randomly, and $t = 0$, Initialize $IL_i^0$ for all agents, $AO_i^t = 0, AS_i^t = 0, \forall i, t$
  **while** $t \leq T$ **do**
    $AO_i^{t+l_i^{fi}} += d^t$ ▷ set the retailer's arriving order to external demand
    **for** $i = 1:4$ **do** ▷ loop through stages downstream to upstream
      get action $a_i^t$ ▷ choose order quantity
      $OO_i^{t+1} = OO_i^t + a_i^t$ ▷ update $OO_i$
      $AO_{i+1}^{t+l_i^{fi}} += a_i^t$ ▷ propagate order upstream
    **end for**
    $AS_4^{t+l_4^{tr}} += a_4^t$ ▷ set manufacturer's arriving shipment to its order quantity
    **for** $i = 4:1$ **do** ▷ loop through stages upstream to downstream
      $IL_i^{t+1} = IL_i^t + AS_i^t$ ▷ receive inbound shipment
      $OO_i^{t+1} -= AS_i^t$ ▷ update $OO_i$
      current_Inv $= \max\{0, IL_i^{t+1}\}$ ▷ determine outbound shipment
      current_BackOrder $= \max\{0, -IL_i^t\}$
      $OS_i^t = \min\{$ current_Inv, current_BackOrder $+ AO_i^t \}$
      $AS_{i-1}^{t+l_i^{tr}} += OS_i^t$ ▷ propagate order downstream
      $IL_i^{t+1} -= AO_i^t$ ▷ update $IL_i$
      $c_i^t = c_i^p \max\{-IL_i^{t+1}, 0\} + c_i^h \max\{IL_i^{t+1}, 0\}$ ▷ calculate cost
    **end for**
    $t += 1$
  **end while**
**end procedure**

## References

L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321, 1992.

J. D. Sterman. Modeling managerial behavior: Misperceptions of feedback in a dynamic decision making experiment. *Management Science*, 35(3):321–339, 1989.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT Press, Cambridge, 1998.