RL-based Approaches for Fulfillment Policies in a Multi-Echelon Inventory System

Xiaohui Tu
11255635
xiaohui.tu@hec.ca

Mingliang Wei
11274244
mingliang.wei@hec.ca

1 INTRODUCTION

1.1 Background

Inventory prediction, which is an important part of the supply chain management, is a difficult part to handle with, since the uncertainty from the customers, uncertainty of lead time are two unpredictable factors. Also, the products distributed from upstream to downstream, the products in transportation and backorder are changing all the time, some traditional methods like Efficient Consumer Response (ECR) cannot give a solution when facing a changing environment.

Although reinforcement learning (RL) is proved to be an effective tool in supply chain management, limited research focuses on finding out adaptive fulfillment policies in a frequently changing multi-echelon inventory system. In line with the trend, this study will use RL-based approaches to find out adaptive fulfillment strategies when facing a changing environment in a multi-echelon inventory system. Two common uncertainties, on lead time and on end-customer demands, will be practically taken into consideration in this study.

1.2 Problem Statement and Research Goals

This research work will employ synthetic data to simulate the uncertainty of demand from customers and the lead time for each echelon, after building the environment of the supply chain system and our RL-based agents, we will find out the approaches to investigate the hidden pattern related to predicting and controlling the multi-echelon inventory system. Our **expected results** can be summarized as follows:

(1) By using the RL-based approaches, we will build a framework used to deal with the multi-echelon inventory problem when facing changing environment.
(2) We aspire to make some contributions to the supply chain field by providing proof-of-concept that reinforcement learning method can help tackle multi-echelon inventory problems in a dynamic setting.
(3) From the learning process, RL-based agents will find out the optimal policies. Obeying these policies, we can provide some insights for inventory predictions. This can be further developed to provide suggestions for managers working in supply chain procurement departments with ordering strategies to help them minimize inventory costs and reduce backlogging orders, thereby increasing their efficiency in supply chain management.

2 LITERATURE REVIEW

In this project, we focus on reinforcement learning methods, the sub-field of machine learning that develops policies for sequential decision-making problems. Despite the ongoing enthusiasm these days for making breakthroughs with machine learning and artificial intelligence, there is still a lack of using RL in industrial environments such as inventory management [7]. Therefore, we mainly review the existing literature in two aspects, namely the study of multi-level supply chain management and the application of reinforcement learning in supply chain management.

2.1 Multi-Echelon Inventory Models

In multi-echelon models, we have multiple stages or echelons that can hold inventory in the centralized or sequential supply chain. Usually, we have to make strong assumptions, such as a serial system or no intermediate terminals [4][6], to reduce the model complexity. And an extensive review of the variety of multi-echelon inventory models studies based on number of echelons, network structure, holding cost functions, etc. are later given by [5].

Our theoretical model is built referring to [10], in which a sequential model with four echelons is adopted and laboratory experiments are used as a research tool to uncover the dynamic and complex nature of continuous replenishment decisions.

2.2 Reinforcement Learning in Supply Chain

Reinforcement Learning are already been widely used in Supply Chain management based on different ordering policies. Considering the inventory level and age of products, stock-based and age-based policies are most common ones used to replenishes stocks [2]. Effectiveness of Case-based

Machine Learning Project, December, 2019, Montreal, CA

ISBN 123-4567-24-567/08/06.

Reinforcement Learning algorithm has be proved in a simplified two-echelon supply chain under time-triggered and event-triggered policy [1].

Bullwhip effect, referring to the phenomenon of amplification of distortion of demand, is very common in multiple members and multi-commodities supply chain, can also be alleviated by using RL algorithm [12].

Not only a centralized supply chain model, RL algorithm can also be used in a decentralized supply chain environment [3]. We will explain more in the following sections to further prove the effectiveness of RL algorithm in a sequential supply chain model.

3 MODEL OF MULTI-ECHELON SUPPLY CHAIN

In this section, we will try to integrate multiple components of the supply chain within a single framework, instead of focusing on one aspect of the supply chain to better understand the inventory replenishment dynamics [11]. Specifically, we build a sequential supply chain model with three echelons (Figure 1), which represents roles of the factory, the wholesaler and the retailer. On the far right, the market demand is unknown and customer orders are transferred in the form of information flows (dotted line in Figure 1) from market to factory, trigging material flows (solid line in Figure 1) from upstream to downstream [10]. More details of the model will be discussed in the next section.

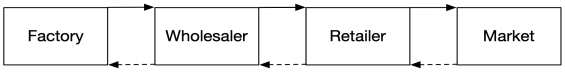

Figure 1: Schematic diagram of a retailer inventory system

3.1 Model Description

Let $i = \{F, W, R\}$ be the echelon index, where F stands for the factory, W for the wholesaler, D for the retailer. $j = M$ is for the market index. Each inventory manager fulfills the orders received from the downstream customer if sufficient on-hand inventory is available, otherwise backlogs are accumulated. Also, each manager has to place order Q_d^i at decision interval T with the upstream supplier. Information flow is assumed to have no delays, but the material flow has an uncertain nominal shipment lead time LT^i following discrete uniform distribution with an interval [1, 4]. The actual shipment lead-time is potentially variable and depends on the availability of on-hand inventory at the upstream supplier. This represents the time required to receive and to ship orders. Therefore, on-hand inventory must be regulated with respect to the shipment delays.

Each echelon is assumed to be information-separate that the downstream and the upstream status are unknown, such as the on-hand inventory level of the upstream supplier, so the inventory replenishment decision at echelon i is made with *local information*, such as the received orders from the downstream customer or the on-hand inventory level.

3.1.1 Reward, State, and Action. From an optimization point of view, the objective is to maximize the total profit (*accumulated reward*) of the chain, namely the accumulated income I minus the accumulated costs C. The income is obtained from multiplying the sale price p by the sale amount q. The costs (*state*) are composed of the ordering cost C_o^i, the holding cost C_i^i, the transportation cost C_p^i and the shortage cost C_d. The sum of these four costs is accumulated at end of each decision interval and used to calculate the accumulated costs C. Also, at each decision point, the order quantities (*action*) are given calling one of the RL-based agents, including the benchmark agent, the SARSA agent, and the DDPQ agent, depending on which agent we specify to use.

The **objective** of our problem can be thus written as:

$$\max \sum P = \sum (I - C) = \sum \{p \cdot q - [\sum (C_o^i + C_i^i + C_p^i + C_d^i)]\}$$

where $i = \{F, W, R\}$ is the echelon index, P is the accumulated profit, I is the accumulated income, C is the accumulated costs, p is the sale price, q is the sale amount, C_o^i is the ordering cost, C_i^i is the holding cost, C_p^i is the transportation cost, and C_d^i is the shortage cost.

3.1.2 Model Assumptions. We mainly use the following assumptions to simplify our model:

(1) information is not shared among echelons, so decisions are made only with local information;
(2) any unfulfilled order is backordered and is fulfilled whenever the material is available in the inventory;
(3) ordering cost is C_o^i when a non-negative order is made, 0 when not ordering;
(4) holding cost at each level is linear to the quantity of inventory;
(5) backlogging cost only occurs at the retailer level and is linear to the quantity of the customer demand.

3.2 Parameter Settings

The following parameters are globally used for the model:

- sale price (dollars): 1000
- costs (dollars)
  - ordering costs: 80
  - holding costs: [3, 5, 10]
  - transportation costs: [3, 5, 10]
  - delaying costs: 50
- supply level (units): infinity
- demand (units): erlang distribution with a mean of 1 and a variance of 1

- safety stock (units): [1, 1, 7]
  - The safety stock is only used for the benchmark agent, which specifies the levels of safety stock for the several down-most nodes such that the last integer of the list is that of the down-most node, the second last integer of the list is that of the second down-most node, etc. Redundant integers will be ignored, and unspecified nodes will have 0 as safety stock.

3.3 Environment

In each step, the following operations are performed:

(1) we first check whether the following *action* inputs are correct: (a) action type (b) shape of the action list (c) action value,
then we reset *reward* and *state*;
(2) In each iteration of the decision interval, we do:
  (a) update costs: the costs are computed according to the *action*, namely how many to order at each echelon, and the state of the chain at last period. The four kinds of costs all need to be updated;
  (b) update orders: each node orders to upstream indicated by the *action*, which is only performed once during the decision period;
  (c) update production: we update the status of the products based on the remaining lead time it has right now. If products have finished the production process, the inventory and transportation status will also be updated for each echelon (here we assume the supplying capability of the factory is infinity).
  (d) update supply: the factory needs a delaying time to send goods to downstream. Nevertheless, the sending information is updated, waiting for the calculation of transportation costs;
  (e) update logistics: transporting information of the wholesaler and the retailer are dealt with in the same way as last step once the goods are received;
  (f) update demand: new demand from the customer is generated and accumulated;
  (g) update trade: when the retailer has available inventory, the transaction is successful. The trading demand depends on the smaller value of the inventory level and the demand;
(3) when the next decision is needed, we return the current state of the chain, including profits already gained, the four types of expenses and income.

4 METHODOLOGY

In this section, we will mainly introduce the two RL-based agents we use for predictions, namely the SARSA(λ) (State–action–reward–state–action lambda) agent based on TD-Learning and the DDPG (deep deterministic policy gradient) agent based on actor-critic. For more valid comparisons, we also established a benchmark agent using the traditional supply chain prediction method, which will be introduced in the following section.

4.1 Benchmark Agent

The benchmark agent is implemented with classical supply chain forecasting models, which orders at each decision point without witnessing the environment and learning from previous experiences.

The order quantity of this model is calculated by adding the *safety stock* mentioned in Section 3.2 to the *target inventory* and subtracting the *inventory level* at the time of ordering.

Further, the *target stock* is obtained by multiplying the *mean* of the demand distribution that we take as the expected demand with *day*, which is the sum of the *time needed* to be covered if we order according to the EOQ model [8] and the specified *lead time*.

The *inventory level* is calculated by adding the *on-hold inventory* to the *upstream order* at the decision time and subtracting the *downstream order* at the decision time.

Although this calculation may seem a little complicated and scary, the model behind follows a rather simple logic and its representative traditional supply chain operations rely heavily on the experience and intuition of practitioners.

4.2 SARSA(λ)

In this section, we will explain the characteristics and key parameters of the Sarsa-lambda algorithm and how it works.

Same as its name, the procedure of the Sarsa algorithm updates Q-values following a sequence of $<S_t, A_t, R_t + 1, S_t + 1, A_t + 1>$. Compared to Q-learning, Sarsa is an on-policy learning algorithm that the maximum reward for the next state s' is not necessarily used for updating Q-values and is therefore more conservative. Based on Sarsa, an eligibility trace table is introduced to record and give different weights to those steps leading to rewards. In this way, the agent will reach convergence quicker than Sarsa algorithm theoretically.

4.2.1 Algorithm Description. As the pseudocode shown in Figure 2, we first initialize the Q-table and the eligibility trace table randomly. In each iteration, an action is selected based on the Q-table using ϵ-greedy, and Q-values are updated according to the reward and the next state. The best action in each state is selected as the optimal ordering policy to do when facing the same state in the future.

The probability of exploration is a fixed number here and we set is as $\epsilon = 0.9$, so in each step, the agent will have

Initialize $Q(s,a)$ arbitrarily and $e(s,a)=0$, for all s,a
Repeat (for each episode):
  Initialize s, a
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    For all s,a:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    $s \leftarrow s'$; $a \leftarrow a'$
  until s is terminal

Figure 2: Sarsa(λ) Algorithm
Source: http://incompleteideas.net/book/first/ebook/node77.html

a 10% chance to choose a random action instead of using the ϵ-greedy to choose the maximum one, the Q-function is estimated by:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))e(s,a)$$

The eligibility trace table is estimated by:

$$e(s,a) = \gamma\lambda e(s,a)$$

By using the eligibility trace table, steps close to high rewards will be strengthened and given higher weights.

The rules mentioned above will be used to update the state-action pair values. In the beginning, since agent has no knowledge of each action in each state, so the initial value of all the Q-functions will be set to 0 for all state-action pairs. During the learning process, the agent will figure out if the current state has been encountered before, if not, a new state-action pair will be created and be updated to the Q-table and eligibility trace table at the same time.

In the learning phase, all the updating rules will be placed in the internal loop and estimated for each state that agents get to. By repeating each state through the simulation period, the value of $Q(s,a)$ for each state-action pair converges. After finishing the learning phase, the best policy in each state will be found out and used in our best policy.

4.2.2 Train and test. Since the Sarsa-lambda algorithm is model-free, we do not have to know how the sequential supply chain model functions. In the training process, the agent gives actions based on the Q-table, then the environment will give a feedback containing new state s', reward r to the agent based on the action it took. The test will be taken in the same environment based on the optimized policy.

4.3 DDPG

The SARSA model has an inherent disadvantage that cannot be overcome in implementation, that is, it depends on the existing experience of Q-table to make decisions, so it cannot handle large-scale problems with high requirements on optimization at each step, and the search processing is slow and low-efficient.

Therefore, we consider to use a new architecture called Actor-Critic with two networks — the actor network and the critic network as shown in Figure 3 to approximate the decision functions.

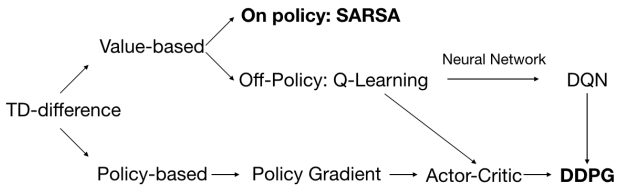

Figure 3: Relationship of the RL algorithms
Source: blog.csdn.net/wordyangl, redrew with minor modifications

More specifically, we choose to use the DDPG method, which is developed on the basis of Deep Q-Network (DQN) method that used to overcome the search and convergence difficulties of Q-table when facing large-scale problems.

4.3.1 Algorithm Basics. Compared to DQN, the DDPG method combines the policy gradient and state action value functions under its Actor-Critic architecture. The implementation details of this algorithm are too complicated and exceed the content learned during this semester. Due to space limitations, we will only introduce this algorithm intuitively from a high level.

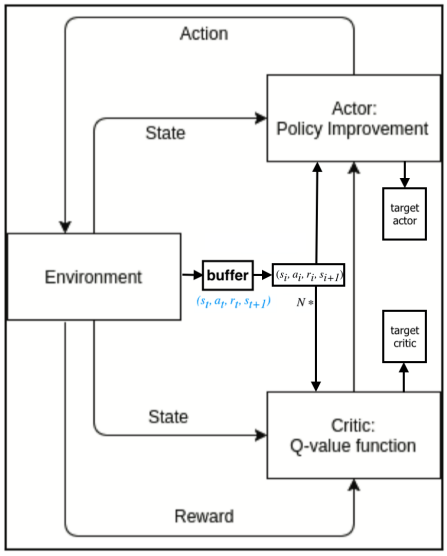

Figure 4: Architecture of the DDPG algorithm
Source: Referenced from [9] with modifications.

As shown in Figure 4, one network acts as a **critic** $Q(s, a; w)$, which evaluates the actor's action by computing the temporal difference error and updates the weight parameter vector w. Then, the other network acts as an **actor** $\mu(s; \theta)$, which performs a policy gradient to select the actions and updates the policy parameter vector θ.

Here, an **experience buffer** is used for which the Actor and Critic networks are trained by sampling a mini batch of experiences. For computing the loss, we also use a separate **target Actor** and **target Critic** networks, which are copies of the online Actor and Critic network.

After some iterations, we slowly update the weights of the target networks with a high discounting level to promote greater stability, which is called the **soft update**. This strategy can help stabilize the state-action updates by avoiding to constantly change the network weights and thus ignore long-term rewards.

4.3.2 Train and Test. This algorithm is model-free, so there is no need to understand the complex mechanism of the supply chain ordering process. At each decision time, we give ordering *actions* through the actor network, and get feedback from the environment, passing the *reward* and *state* to the critic network. The **training** is to update the neural network parameters of the actor and the critic as introduced in last section. We then **test** the algorithm in the same environment, and during the test, the neural networks will continue learning.

5 COMPUTATIONAL EXPERIMENTS

5.1 Experimental Setup

The sequential supply chain model and the two algorithms were implemented in Python 3.7 with the use of its scientific libraries, and the algorithms additionally was developed with the newly released Tensorflow 2.0.0. This is an open-source Python library for machine learning in a variety of perception and language understanding tasks, making it a strong and easy tool for developers to save time on writing codes.

One computer with 8 CPUs, 30G of RAM, and 2 Tesla V100 GPUs is used for running the Sarsa-Lambda algorithm. The other computer with 6 cores operating at 2.2 GHz (up to 4.1 GHz with turbo boost) and 16GB of RAM was used for testing the DDPG algorithm.

5.2 Experiments on SARSA(λ)

5.2.1 Parameter Settings. During the pre-experiment, we found that Sarsa-lambda run very slowly. We thus had to choose relatively small parameters as follows:

- lead times: discrete uniform distribution [1, 2]
- decision interval: 1 days
- maximal action: 3
- trace decay rate(λ): 0.3/0.6/0.9

The first reason for changing parameter settings is that if we choose the maximum action of 30 as DDPG, the action list for the agent will be huge. Also, if we have a high value of the decision interval, it means that the backorders will be more likely to be accumulated and lead to more extreme situation and a larger state set. All of the reasons mentioned above will make the Q-table too large and increase the calculating time, making it more difficult for the agent to converge. Therefore, the parameters listed above, as well as a constant customer demand, are used in the SARSA(λ) agent. This will help prevent the agent from meeting new state in every step as the demands of the customer are fixed in this simplified situation. As a result, instead of maximizing the profit, our objective function here is to minimize the total cost.

5.2.2 Model Training. By training the Sarsa-lambda agent with 100 episodes and with an episode length of 200 days, we can get our accumulated rewards as shown in Figure 5.

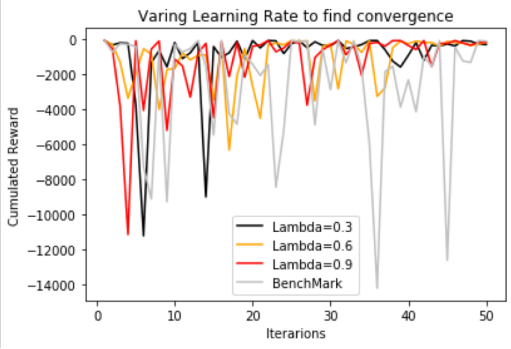

Figure 5: Accumulated reward

From the training process, we can see that the SARSA method is better than the benchmark in almost all situations and it gradually converges under all settings of the trace decay rate.

By tuning the trace decay rate λ, we can find out the best value here is λ=0.9, in which situation the agent converges quickest. We will further test the trained model in the same environment in the next section.

5.2.3 Model Performance. Finally, we tested our Sarsa-lambda agent using λ of 0.9 with 50 episodes, each with a length of 200 days. The accumulated rewards are shown in Figure 6.

Since costs are used as the objective value, a summed reward of 0 is already the best case. From the testing process, we can see that the SARSA method is better than the benchmark in almost all cases, but it does not show an absolute advantage. Moreover, this method has simplified the model to some extent, and its validity needs to be further verified before using for practice.

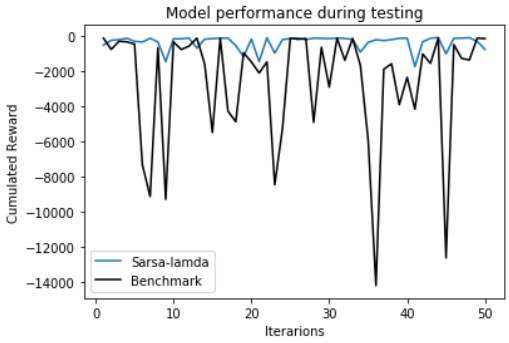

Figure 6: Model Performance of Sarsa-lambda during testing

5.3 Experiments on DDPG

5.3.1 Parameter Settings. We tested the DDPG algorithm in the above experimental environment. The parameters different from the SARSA algorithm are listed as follows:

- lead times: discrete uniform distribution [1, 4]
- decision interval: 10 days
- maximal action: 30

5.3.2 Model Training. We trained our DDPG agent with 100 episodes, each with a length of 400 days. The training loss and td errors are shown on the left of Figure 7, and the accumulated rewards on the right.

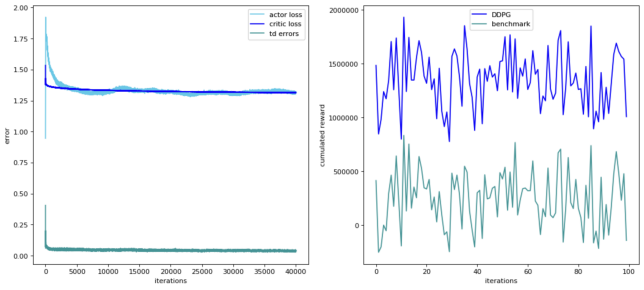

Figure 7: Model performance of DDPG during training

From the training process, we can see that the parameters of the DDPG method are updated well soon, and the loss continues to gradually decrease during training. However, the rewards are not significantly risen as expected, which implies that the predictions of the DDPG agent is still more like a random behavior. But it is worth noting that DDPG agent still shows absolute advantages compared to benchmark.

5.3.3 Model Performance. Finally, we tested our DDPG agent with 50 episodes, each with a length of 800 days. The loss and td errors are shown on the left of Figure 8, and the accumulated rewards on the right.

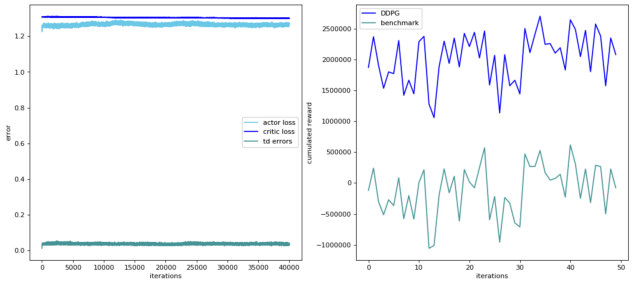

Figure 8: Model performance of DDPG during testing

From the testing process, we can see that the parameter updates of the neural networks have been relatively stable. Although the return of rewards still fluctuates, it shows an upward trend at the end. At the same time, the DDPG agent is still much better than the benchmark results, proving the effectiveness of this algorithm.

6 CONCLUSIONS AND FUTURE DIRECTIONS

6.1 Conclusions

In this work, we built a three-echelon supply chain decision model and implemented two reinforcement learning algorithms to assist decision making on ordering quantities at each decision points. A benchmark agent following the traditional purchasing practices is also used to compare and valid the performance of two models. Finally, we can reach the following conclusions:

First, both the Sarsa(λ) and the DDPG agents are proved to be better than the Benchmark agent, suggesting the effectiveness of these two algorithms in tackling the given specific dynamic inventory fulfillment problems.

However, we should not ignore that Sarsa(λ) is not so effective as the running time of each round during training is about 20 minutes, while DDPG only takes about 50 seconds. Also, this method over-simplifies the model. When facing a more sophisticated environment with a large number of actions and states, Sarsa(λ) has an inner deficiency in optimization while DDPG outperforms.

Still, DDPG needs to be further improved. As we can see, errors of the DDPG decrease fast and the network generalizes well with the current policy, but the rewards are expected to increase more significantly. This might be due to the fact that policy has settled into a pattern where values can be estimated well by the neural network, but agent still does not find improvements to that policy. Changing the hyperparameters of the neural network or lengthening the training process may help the convergence of policies. Nevertheless, it should be noted that even in such a simplified supply chain model, there are too many parameters and too many sources

of uncertainty, making this problem both practically and theoretically difficult.

Finally, we have realized that parameter tuning is effort-intensive and an art of science, and we are left to explore how to cover better policies.

6.2 Limitations and Future Directions

There are still many limitations in this work that we should pay attention to.

First, even if we use smaller discrete ranges on SARSA to make the computing affordable, we may lose important features and end up with a huge set of action spaces. When we have a huge space for action, it is naturally difficult to achieve convergence. If we insist on using this method, we need to improve searching strategy on Q-table to make it more effective.

Also, our model is actually a simplified one such that the assumption on information isolation is not fully achievable in the actual situations. This also makes the effectiveness of the ordering strategy questionable. Eventually, good algorithms have to be good ordering decisions. We hope that in the future, we can use real data and more complex situations to test our algorithms. Still, this project is an interesting initial attempt for further explorations.

ACKNOWLEDGMENTS

Thanks for the opportunity given by Professor Laurent Charlin of this course project to help us practice and integrate what we have learnt in class. Inspired by [7], we had planned to implement the A3C algorithm at the beginning, but suspended due to its difficulty. The DDPG was then developed instead and the ideas are still unpolished. This project is developed during this semester, with parts of this research work under the financial supporting from GERAD that Xiaohui wants to express sincere thanks to.

REFERENCES

[1] 2009. Case-based reinforcement learning for dynamic inventory control in a multi-agent supply-chain system. *Expert Systems with Applications* 36, 3, Part 2 (2009), 6520 – 6526. https://doi.org/10.1016/j.eswa.2008.07.036

[2] 2018. Reinforcement learning approaches for specifying ordering policies of perishable inventory systems. *Expert Systems with Applications* 91 (2018), 150 – 158. https://doi.org/10.1016/j.eswa.2017.08.046

[3] S. Kamal Chaharsooghi, Jafar Heydari, and S. Hessameddin Zegordi. 2008. A reinforcement learning model for supply chain ordering management: An application to the beer game. *Decision Support Systems* 45, 4 (2008), 949 – 959. https://doi.org/10.1016/j.dss.2008.03.007

[4] Andrew J Clark and Herbert Scarf. 1960. Optimal policies for a multi-echelon inventory problem. *Management science* 6, 4 (1960), 475–490.

[5] Ton de Kok, Christopher Grob, Marco Laumanns, Stefan Minner, Jörg Rambau, and Konrad Schade. 2018. A typology and literature review on stochastic multi-echelon inventory models. *European Journal of Operational Research* 269, 3 (2018), 955–983.

[6] Awi Federgruen and Paul Zipkin. 1984. Approximations of dynamic, multilocation production and inventory problems. *Management Science* 30, 1 (1984), 69–84.

[7] Joren Gijsbrechts, Robert N Boute, Jan A Van Mieghem, and Dennis J Zhang. [n.d.]. Can Deep Reinforcement Learning Improve Inventory Management? Performance on Dual Sourcing, Lost Sales and Multi-Echelon Problems. ([n. d.]).

[8] M Goh. 1994. EOQ models with general demand and holding cost functions. *European Journal of Operational Research* 73, 1 (1994), 50–54.

[9] Arthur Guez Hado Van Hasselt and David Silver. [n.d.]. Deep Reinforcement Learning with Double Q-Learning. ([n. d.]).

[10] Amin Kaboli, Rémy Glardon, Nicolas Zufferey, and Naoufel Cheikhrouhou. 2019. Replenishment behaviour in sequential supply chains. *International Journal of Logistics Systems and Management* 32, 3-4 (2019), 322–345.

[11] Benjamin Van Roy, Dimitri P Bertsekas, Yuchun Lee, and John N Tsitsiklis. 1997. A neuro-dynamic programming approach to retailer inventory management. In *Proceedings of the 36th IEEE Conference on Decision and Control*, Vol. 4. IEEE, 4052–4057.

[12] Gang Zhao and Ruoying Sun. 2010. Application of multi-agent reinforcement learning to supply chain ordering management. In *2010 Sixth International Conference on Natural Computation*, Vol. 7. IEEE, 3830–3834.