Titre: Title:	Deep Learning and Reinforcement Learning for Inventory Control
Auteur: Author:	Zahra Khanidahaj
Date:	2018
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Khanidahaj, Z. (2018). Deep Learning and Reinforcement Learning for Inventory Control [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/3844/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/3844/
Directeurs de recherche: Advisors:	Louis-Martin Rousseau
Programme: Program:	Maîtrise recherche en génie industriel

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Deep Learning and Reinforcement Learning for Inventory Control

ZAHRA KHANIDAHAJ

Département de mathématiques et de génie industriel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

Génie industriel

Décembre 2018

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

Deep Learning and Reinforcement Learning for Inventory Control

présenté par **Zahra KHANIDAHAJ**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

a été dûment accepté par le jury d'examen constitué de :

Michel GENDREAU, président

Louis-Martin ROUSSEAU, membre et directeur de recherche

Andrea LODI, membre

DEDICATION

To my parents

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor, Professor Louis-Martin Rousseau, for his valuable guidance and continuous technical and financial supports during my research. In addition, I express my sincere gratitude to Dr. Yossiri Adulyasak, for his technical support during my project.

Also, I would like to express my appreciation to my thesis committee members, Professor Michel Gendreau and Professor Andrea Lodi for their time and constructive comments. In Addition, I appreciate my intimate friends for their helps during this period.

Many thanks go to Dr. Joelle Pineau and Dr. Doina Precup at McGill University. I augmented my knowledge about deep learning and reinforcement learning considerably by the valuable comments and discussions during their courses.

And last but not least, special thanks to my beloved family. I cannot find any word expressing my deepest thanks and appreciation. I am highly in my dear family's debt for their endless support and unconditional love and encouragement even I am thousands of kilometres far from home. They have always been helping me to pursing my goals and been supporting me continuously, throughout my life.

RÉSUMÉ

La gestion d’inventaire est l’un des problèmes les plus importants dans la fabrication de produits. Les décisions de commande sont prises par des agents qui observent les demandes, stochastiques, ainsi que les informations locales tels que le niveau d’inventaire afin de prendre des décisions sur les prochaines valeurs de commande. Étant donné que l’inventaire sur place (la quantité disponible de stock en inventaire), les demandes non satisfaites (commandes en attente), et l’existence de commander sont coûteux, le problème d’optimisation est conçu afin de minimiser les coûts. Par conséquent, la fonction objective est de réduire le coût à long terme) dont les composantes sont des inventaires en stock, commandes en attente linéaires (pénalité), et des coûts de commandes fixes.

Généralement, des algorithmes de processus de décision markovien, et de la programmation dynamique, ont été utilisés afin de résoudre le problème de contrôle d’inventaire. Ces algorithmes ont quelques désavantages. Ils sont conçus pour un environnement avec des informations disponibles, telles que la capacité de stockage ou elles imposent des limitations sur le nombre d’états. Résultat, les algorithmes du processus de décision markovien, et de la programmation dynamique sont inadéquats pour les situations mentionnées ci hauts, à cause de de la croissance exponentielle de l’espace d’état. En plus, les plus fameuses politique de getsion d’inventaire, telles que politiques standards <*s*,*S*> et <*R*,*Q*> ne fonctionne que dans les systèmes où les demandes d’entrées obtiennent une distribution statistique connues.

Afin de résoudre le problème, un apprentissage par renforcement approximée est développé dans le but d’éviter les défaillances mentionnées ci hauts. Ce projet applique une technique d’apprentissage de machine nommé ‘Deep Q-learning’, qui est capable d’apprendre des politiques de contrôle en utilisant directement le ‘end-to-end RL’, malgré le nombre énorme d’états. Aussi, le modèle est un ‘Deep Neural Network’ (*DNN*), formé avec une variante de ‘Q-learning’, dont l’entrée et la sortie sont l’information locale d’inventaire et la fonction de valeur utilisée pour estimer les récompenses futures, respectivement.

Le Deep Q-learning, qui s’appelle ‘Deep Q-Network’ (*DQN*), est l’une des techniques pionnières ‘*DRL*’ qui inclut une approche à base de simulation dans laquelle les approximations d’actions sont menées en utilisant un réseau *DNN*. Le système prend des décisions sur les valeurs de commande. Étant donnée que la fonction de coût est calculée selon l’ordre *‘O’* et le niveau d’inventaire *‘IL’*, les valeurs desquelles sont affectées par la demande *‘D’*, la demande d’entrée ainsi que l’ordre et le niveau d’inventaire peuvent être considérés en tant qu’information individuelle d’inventaire. De plus, il y a un délai de mise en œuvre exprimant la latence dans l’envoi des informations et dans la réception des commandes. Le délai de mise en œuvre fournit davantage d’information locale incluant *‘IT’* et *‘OO’*. Le *‘IT’* et *‘OO’* sont calculés et suivis durant les périodes de temps différents afin d’explorer plus d’informations sur l’environnement de l’agent d’inventaire. Par ailleurs, la principale information individuelle et la demande correspondante comprennent les états d’agents.

Les systèmes *‘PO’* sont davantage observés dans les modèles à étapes multiples dont les agents peuvent ne pas être au courant de l’information individuelle des autres agents. Dans le but de créer une approche basée sur le ‘*ML*’ et fournir quelques aperçus dans la manière de résoudre le type d’agent multiple *‘PO’* du problème actuel de contrôle d’inventaire, un agent simple est étudié. Cet un agent examine si on peut mettre sur pied une technique ‘*ML*’ basée sur le ‘*DL*’ afin d’aider à trouver une décision de valeur de commande quasi optimale basée sur la demande et information individuelle sur une période à long terme. Afin de le réaliser, dans un premier temps, la différence entre la valeur de commande (action) et la demande comme résultat d’un ‘*DNN*’ est estimée. Ensuite, la commande est mise à jour basée sur la commande à jour et la demande suivante. Enfin, le coût total (récompense cumulative) dans chaque étape de temps est mis à jour. En conséquence, résoudre le problème de valeur de commande d’agent simple suffit pour diminuer le coût total sur le long terme. Le modèle développé est validé à l’aide de différents ratios des coefficients de coût. Aussi, le rendement de la présente méthode est considéré satisfaisant en comparaison avec le ‘*RRL*’ (*RL* de régression), la politique <*R,Q*> et le politique <*s,S*>. Le *RL* de régression n’est pas capable d’apprendre aussi bien et avec autant de précision que le ‘*DQN*’. En dernier lieu, des recherches supplémentaires peuvent être menées afin d’observer les réseaux de chaînes d’approvisionnement multi-agents en série partiellement observables.

ABSTRACT

Inventory control is one of the most significant problems in product manufacturing. A decision maker (agent) observes the random stochastic demands and local information of inventory such as inventory levels as its inputs to make decisions about the next ordering values as its actions. Since inventory on-hand (the available amount of stock in inventory), unmet demands (backorders), and the existence of ordering are costly, the optimization problem is designed to minimize the cost. As a result, the objective function is to reduce the long-run cost (cumulative reward) whose components are linear holding, linear backorder (penalty), and fixed ordering costs.

Generally, Markov Decision Process (*MDP*) and Dynamic Programming (*DP*) algorithms have been utilized to solve the inventory control problem. These algorithms have some drawbacks. They are designed for the environment with available local information such as holding capacity or they impose limitations on the number of the states while these information and limitations are not available in some cases such as Partially Observable (*PO*) environments. As a result, *DP* or *MDP* algorithms are not suitable for the above-mentioned conditions due to the enormity of the state spaces. In addition, the most famous inventory management policies such as normal <*s,S*> and <*R,Q*> policies are desirable only for the systems whose input demands obtain normal distribution.

To solve the problem, an approximate Reinforcement Learning (*RL*) is developed so as to avoid having the afore-mentioned shortcomings. This project applies a Machine Leaning (*ML*) technique termed Deep Q-learning, which is able to learn control policies directly using end-to-end RL, even though the number of states is enormous. Also, the model is a Deep Neural Network (*DNN*), trained with a variant of Q-learning, whose input and output are the local information of inventory and the value function utilized to estimate future rewards, respectively.

Deep Q-learning, which is also called Deep Q-Network *(DQN),* is one of the types of the pioneer Deep Reinforcement Learning (*DRL*) techniques that includes a simulation-based approach in which the action approximations are carried out using a Deep Neural Network (*DNN*). To end this, the agents observe the random stochastic demands and make decisions about the ordering values. Since the cost function is calculated in terms of Order (*O*) and Inventory Level (*IL*) whose values are affected by Demand (*D*), input demand as well as the order and inventory level can be considered as the individual information of the inventory. Also, there is a lead-time expressing the latency on sending information or receiving orders. The lead-time provides more local information including Inventory Transit (*IT*) and On-Order (*OO*). *IT* and *OO* are calculated and tracked during different time periods so as to explore more information about the environment of the inventory agent. Furthermore, the main individual information and the corresponding demand comprise the states of the agent.

PO systems are observed more in multi-stage models whose agents can be unaware of the individual information of the other agents. In order to create a *ML*-based approach and provide some insight into how to resolve the *PO* multi-agent type of the present inventory control problem, a single-agent is studied. This agent examines if one can implement a *ML* technique based on Deep Learning (*DL*) to assist to learn near-optimal ordering value decision based on demand and individual information over long-run time. To achieve this, first, the difference between the ordering value (action) and demand as the output of a *DNN* is approximated. Then, the order is updated after observing the next demand. Next, the main individual information of the agent called input features of a *DNN* is updated based on the updated order and the following demand. Lastly, the total cost (cumulative reward) in each time step is updated. Accordingly, solving the ordering value problem of single-agent suffices to diminish the total cost over long-run time. The developed model is validated using different ratios of the cost coefficients. Also, the performance of the present method is found to be satisfactory in comparison with Regression Reinforcement Learning (Regression *RL*), <*R,Q*> policy, and <*s,S*> policy. The regression *RL* is not able to learn as well and accurately as *DQN*. Finally, further research can be directed to solve the partial-observable multi-agent supply chain networks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

ADAM	ADAptive Moment estimation
AI	Artificial Intelligence
BP	Back Propagation
DL	Deep Learning
DNN	Deep Neural Network
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
ER	Experience Replay
FC	Fully Connected
IID	Independent and Identically Distributed
MAE	Mean Absolute Error
MDP	Markov Decision Process
ML	Machine Learning
MSE	Mean Square Error
NN	Neural Network
POMDP	Partial Observable Markov Decision Process
PReLU	Parametric Rectified Linear Unit
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
SARSA	State-Action-Reward-State-Action
SGD	Stochastic Gradient Descent
TD	Temporal Difference

CHAPTER 1 INTRODUCTION

1.1 Motivation and Objective

Inventory control is a well-known problem in the field of product manufacturing. Inventory controller (agent) decides about the ordering value based on the input demand in order to reduce the long-run total system cost consisting of linear holding, backorder, and fixed ordering costs. The unpredictability nature of the demand, which is due to its dynamic and random property, makes it reasonable to obtain a new approach to solve the inventory control problem even though there is a number of inventory models. Therefore, information-based decision making (using agents) is desirable. The current agent-based solutions induce some limitations on the values of data, which is not favorable generally. Accordingly, in the present research, a type of Deep Reinforcement Learning (*DRL*) method called Deep Q-Networks (*DQN*) is utilized to solve the inventory control problem.

A question can be raised why *DRL* is preferred to the other methods such as Dynamic Programming (*DP*), Reinforcement Learning (*RL*), and Deep Learning (*DL*). To answer this question, a detailed discussion based on the previous research works by Lin (1993), Van Roy et al. (1998), Mnih et al. (2013), Mnih et al. (2015), Van der Pol and Oliehoek (2017), and Sutton and Barto (2018) is presented in this section. In addition, a number of RL approaches for inventory control are described in Chapter 2. *DP* is inapplicable in most real problems because it is computationally very expensive. Also, most of the *RL* methods impose some limitations or require some pre-knowledge, which are not generally applicable. As a result, one of the long-term challenges of *RL* is to be able to learn how to control the agents directly from enormous inputs, similar to speech recognition. Most of the prosperous *RL* applications utilize hand-made features together with linear value functions or policy representation. Therefore, their performance is highly dependent on how good the features are. The advancement in DL makes the extraction of high-level features from raw datasets possible in some fields such as speech recognition. These approaches employed a type of *DNN* and both Supervised Learning (*SL*) and unsupervised learning. A comparative study of capabilities and incapabilities of *RL*, *DL*, and *SL* is presented herein. A *RL* method faces some challenges from a *DL* approach standpoint. For instance, *DL* techniques are applicable if a large value of labelled training data are available. However, *RL* approaches should be able to learn from a frequently sparse, noisy, and delayed scalar reward. A large delay in observing the effect of an action on the reward is a negative point especially in comparison with the direct input-output relation in *SL*. Another challenge is that most *DL* approaches consider the independent data samplings, whereas *RL* techniques face sequences of much correlated states. In addition, the data distribution in a number of *RL* methods changes with the new behaviors learnt by the algorithm, whereas this can be a challenge for *DL* methods in which the data distribution is considered to be constant. The present research shows that a *DNN* can tackle the aforementioned problems so as to learn appropriate policies from raw datasets in complicated *RL* systems. To end this, a variant of the Q-learning method (Watkins and Dayan 1992) are utilized to train *DNN* using an optimizer for weight updates. The data correlation and non-stationary input distribution issues are mitigated by using Experience Replay (*ER*) sampling from the previous transitions at random, which makes the training distribution more accurate and smoother.

A Markov decision process (*MDP*) is a discrete time stochastic control process made of states, actions, rewards, and transition probabilities. Despite the fact that the problem with single product, single-stage, and a limited number of states (limitations on individual parameters and inputs) can be solved using *MDP*, the present research work is aimed at exploring a type of *DRL* approaches called *DQN*, obtaining some insights and examining the possibility of proper learning of the ordering value when there is no pre-knowledge or limitation on local information such as inventory capacity. This means that to reduce the long-run overall system cost, the stochastic random demands are the inputs, which affect the *RL* algorithm (a variant of Q-learning) whose actions are the ordering values approximated with the assistance of a *DL*. The inputs of this *DNN* structure are the important parameters (features) of inventory control including inventory level, inventory transit (inventory received in transit), on-order inventory (inventory sent but not received yet), ordering, and stochastic demand of the current time. Also, the output of *DNN* is the difference between the next order and the next demand (i.e. X=O-D). Since the stochastic demand of the next time is available as the input (D), the greedy calculation of the ordering value is conducted to be used by the ϵ-greedy rule. In addition, another input is lead-time (*LT*) which is equal to two. Consequently, the next time values of the other features and parameters such as inventory level (*IL*), inventory transit (*IT*), and on-order (*OO*) are calculated by the formulae presenting the relations between the different parameters on different time steps (They are explained in more details in Chapter 4). Therefore, the updated versions of *IL*, *IT*, *OO*, *O*, and *D* are the next inputs while approximation of the difference between the next order and next demand (*X*=*O*-*D*) is the output of the next time of *DNN* (see Tables 1.2 and 1.3). The algorithm can be utilized without any limitation on some parameters such as inventory level, linear holding cost, linear backorder cost, different lead-time values, and type of demand distribution if it is determined.

1.2 Problem Statement

It is essential to gain the sufficient knowledge related to inventory control cost so as to respond to the inventory challenges. Tracking the inventory level (even positive or negative) and the number of times of ordering are unavoidable aspects of a successful inventory management in order to minimize the long-run total system cost. The components of this cost are linear holding, linear backorder (penalty), and fixed ordering costs associated with positive inventory level, negative inventory level, and the times of orders, respectively. The ordering value should be set to a near-optimal value so that the large number of times of ordering and the large values of holdings and backorders are avoided. In product manufacturing, the process of tracking incoming and outgoing goods (orders and demands) is called inventory management. The inventory management is investigated by an agent which makes decision about new orders (actions). This process is conducted after observing the stochastic input demands and by considering the inventory parameters in order to reduce the total cost (the long-run system cost).

Since there are some relations between components of individual information of the agent such as inventory transit, on-order value, and inventory level and the corresponding demand and order, in each time step, their next value is determined, the cumulative reward is updated, and the process continues until the last running time step. It should be mentioned that since the present solution is based on RL, “reward” is used instead of “cost” and their concepts are the same in this research.

Demand is observed as the input of inventory agent, while inventory controller makes decision about the order which is sent to the environment as its output (Figure 1.1). This decision is made by considering not only demand and order but also individual information of inventory such as inventory level. To capture the near-optimal overall cost of inventory, appropriate orders should be found as the outputs of inventory management.

Demand
Inventory
Order

Figure 1.1 Input and output on single-stage inventory problem

In order to solve this inventory control problem, an approach based on a combination of *DL* and *RL* is implemented so as to reduce the cumulative cost of an inventory agent which is executed on a long-run time. This aim is realized using a *RL* algorithm in which the difference between the action (order) and input (demand) is learned by a *DNN*. If the demand and order of each time step are considered as the parts of the individual information (state) of the decision maker (agent), the state of RL algorithm is the input of the *DL* section. The validation of the proposed technique is examined by comparing some methods such as <*s,S*> policy, <*R,Q*> policy and the regression *RL* approach.

In addition, each agent refers to one-stage decision maker in the inventory control optimization problem. There is only one type of ordering product in this research, while the algorithm works for multi-product environment whose products are independent from each other. This research project is aimed at finding the near-optimal overall cost of single-agent (single-stage) when the inventory agent faces the stochastic demands *D* as the input of the environment during the long-run time periods *T*. This optimization is performed by making decisions about the ordering value *O* of each time step. If IL shows inventory level, linear cost for holding (if $IL>0$) and backorder inventory cost (if $IL<0$), and fixed cost for ordering value (if *O*>*0*) are considered in the cost function while their cost coefficients are C_h, C_p, and C_o, respectively.

The general mechanism for the sequence of events including arriving pipeline order, observing the system state, making decision about the order, observing demand, and updating the cost, is shown in Figure 1.2. In addition, in each time step t of a serial multi-agent system, arriving pipeline order illustrates the demand requested from agent i, which is equal to the order of the previous agent $i-1$ with a latency of lead-time LT. This mean that $D_t^i = O_{t-LT}^{i-1}$, if the retailer is the first agent. More details about simulating the environment including different parameters and their relations are given in Tables 1.2 and 1.3 and Chapter 4.

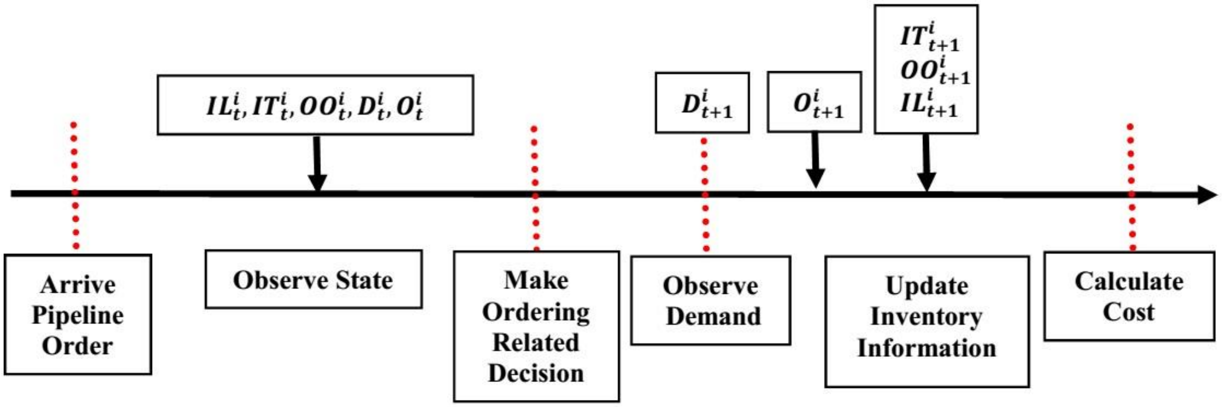

Figure 1.2 The general mechanism for the sequence of events

1.2.1 Type of Inventory Model

Table 1.1 displays the different types of the inventory models. The names of the settings of the models used in the research are made bold and italic. As shown in the table, demand is stochastic, which is selected randomly among one, two, and three, and lead-time is equal to two. Time horizon is set to 500, there is one product, unmet demands are allowed, and there is no limit to capacity. The unrestricted capacity of inventory is one of the benefits of the present research in comparison with the past *MDP* approaches. The time horizon is reasonably high and its value is chosen by considering working 5 days in every working week for two years (or two times (morning and evening) in every working day of a year). It is assumed that the system does not work for two weeks due to the New Year holiday. The demand and lead-time are the independent

Table 1.1 Types of inventory model (The methods written in bold, italic format are used in the present research)

Parameters	**Type**	**Type**	**Type**
Demand	Constant	Deterministic	***Stochastic-random(1-3)***
Lead-Time	“0”	***“>0” -LT=2***	Stochastic
Horizon	Single Period	***Finite (T=500)***	Infinite
Products	***One Product***	Multiple Products	-
Capacity	Order/Inventory Limits	***No Limits***	-
Service	Meet All Demand	***Shortages Allowed***	-

parameters coming from the environments and are the real inputs of *DQN*. However, since the lead-time is constant, it is not considered as an input parameter. *DQN* makes decision about ordering-related values as its outputs. The demand is stochastic and is selected randomly among one, two, and three, while lead-time is assumed to be constant equal to two.

1.2.2 DL and RL Components of DRL

In this inventory control problem, the near-optimal long-run cost function consisting of the linear holding, linear backorder, and fixed ordering costs is obtained. This optimization is found by making decision about the ordering value. The inputs are demands during different time steps and the actions are ordering values. The algorithm utilized in this research project is *DQN*, whose *RL* algorithm is Q-learning. Since there is a huge possibility for different pairs of inputs (demands) and actions (orders), it is impossible to obtain a complete prepared Q-table. Therefore, action selection based on inputs is an online approximation process. The actions of RL are approximated with a *DNN*. The output of *DL* is the difference between orders and demands.

Since the present solution is based on *RL*, “reward” is used instead of “cost” and their concepts are the same in this research. The general goal of *DQN* is to reduce the long-term cumulative

Table 1.2 Components of inventory optimization problem for agent *i*, time *t* *(i=1, t<T, T=500)*

Section	**Detail**	
Cost (reward)	Cost (Reward)	$\mathrm{Cost}_t^i = C_h.\ \mathrm{IL}_t^{i\,+} + C_p.\ \mathrm{IL}_t^{i\,-}$ $\mathrm{Cost}_t^i = \mathrm{Cost}_t^i + C_o\ \text{ if }\ O_t^i > 0$
Input	Demand	$D_t^i = 1, 2, 3$ (Dynamic Input)
	Lead-Time	$\mathrm{LT} = 2$ (Static Input)
Decision Variable	Ordering Value	O_t^i
	Inventory Level	IL_t^i
	On-Order Inventory	OO_t^i
	Inventory Transit	IT_t^i

cost (reward) of the agent where the cost of the agent at each time is defined in Table 1.2. The relations between the parameters are presented in Chapter 4. Also, Table 1.2 defines the inventory optimization problem, while Table 1.3 illustrates input/output of *DRL* as well as the different sections of *DL* and *RL*. Since the *RL* makes decision about ordering, it is a Reinforcement Learning Ordering Mechanism (*RLOM*). The reward function, state, and action are three main components determining *RL* and given in Table 1.3. For instance, inventory level is one of the components of each state of *RL* algorithm, while it is one of the parts of each input of *DL*. It should be mentioned that although the lead-time is equal to a constant value in all the case studies under study in this research, it can be any positive integer. Since the values of parameters are related to the previous time steps, it will be shown that instead of considering one time step of each parameter, a frame with size *k* of the parameters gives the real ones. The details

Table 1.3 Different sections of *DRL* and its *RL* and *DL* sections for agent *i*, time *t* *(i=1, t<T, T=500)*

Section	***DL/RL/DRL***	**Detail**	
Reward (cost)	*RL*	Reward (Cost)	$\mathrm{Cost}_t^i = C_h.\ {IL_t^i}^+ + C_p.\ {IL_t^i}^-$ $\mathrm{Cost}_t^i = \mathrm{Cost}_t^i + C_o$ if $O_t^i > 0$
Input Parameters	*DRL*	Demand	D_t^i (uniform (1-3))
	DRL	Lead-Time	$LT = 2$
Output	*DRL*	Ordering Value	O_{t+1}^i
Type of Agent	*RL*	*RLOM* (*RL* Ordering Mechanism)	-
Algorithm	*RL*	Variant of Q-learning	-
Input/State (Observation)	*DL/RL*	Demand	D_t^i
	DL/RL	Ordering Value	O_t^i
	DL/RL	Inventory Level	IL_t^i
	DL/RL	On-Order Inventory	OO_t^i
	DL/RL	Inventory Transit	IT_t^i
Output/Action	*DL/*	(Order-Demand)/	$X_{t+1}^i = O_{t+1}^i - D_{t+1}^i$/
	RL	Ordering Value	O_{t+1}^i

and relations between decision variables are described in Chapter 4. All of the parameters in Tables 1.2 and 1.3 are previously defined. In addition, ${IL_t^i}^+/{IL_t^i}^-$shows the inventory level if it is larger/lower than zero and the absolute value of IL_t^i is considered in the cost function. The formulae of updating the dependent parameters are presented in the following chapters.

1.3 Contributions

The contributions of the present research work are listed as follows:

The algorithm is designed for an unlimited range of the values of the individual information, while as far as the literature reveals, there are some limitations on the range of individual information such as inventory level for most of the available *MDP* models. In addition, there is no need to know the input distribution, whereas in the most of the previous works, the demand distribution is required to be known a priori. Also, lead-time related parameters such as on-order inventory and inventory transit are considered. Moreover, the influences of different values of hyperparameters on the performance are examined. Finally, the performance of *DQN* method is compared with <*s,S*> and <*R,Q*> policies and linear regression RL method.

1.4 Thesis Structure

This thesis comprises five chapters which are briefly described below:

Chapter 1 explains the motivation and objectives, problem statement, and major contributions of the present research and outlines the thesis scope. Chapter 2 is devoted to reviewing previous studies for inventory control. Chapter 3 describes theory and formulations related to the research area of inventory management. Chapter 4 is allocated to the proposed methodology adopted herein to solve the problem. It also discusses the results. Finally, Chapter 5 presents a summary and the main conclusions of the present research. It also proposes some suggestions for future works in this filed.

CHAPTER 2 A REVIEW OF INVENTORY CONTROL

A *MDP* is a formal way to describe the sequential decision-making problems observed in *RL*. *MDP* is not only tractable to solve but is also relatively easy to specify as it assumes to have perfect knowledge of state. All required information to complete the final task is available in fully observable environments. On the other hand, Partially-Observable Markov Decision Processes (*POMDP*) act uniformly with all sources of uncertainty. Information gathering actions are permitted in *POMDP* and yet solving the problem optimally is often highly intractable.

In the field of inventory optimization, there is a number of research works based on a *MDP*. Van Roy et al. (1997) presented a viable approach based on Neuro-Dynamic Programming (*NDP*) to solve inventory optimization including a retailer. They formulated two dynamic programming studies containing 33 and 46 state variables. Since the state-space of DP models was large, they could not apply classical *DP* approaches. Therefore, they implemented the approximate dynamic programming method to simulate this approximation with a Neural Network (*NN*). Their method falls into the class of *NDP* techniques. The efficiency of their results was assessed by comparing to S-type policies. Moreover, they examined the reduction in the average inventory cost. The results showed that their optimal control technique provided a reward of about ten percent lower than the reward obtained by heuristic methods. Their research has several restrictions on some parameters such as the number of states and the capacity of inventory. Also, Sui et al. (2010) proposed a *RL* approach to find a replenishment policy in a vendor management inventory system with consignment inventory. They did not consider the ordering cost and also divided the state space into 50 regions. In contrast, in this research, the ordering cost is included and the real state is studied.

There is a number of *RL* research studies in the field of inventory control designed for the beer game, which is a serial supply chain network containing (mostly) four agents (stages). The game has a multi-agent, decentralized, independent learner, and cooperative Artificial Intelligence (AI) environment considering holding and back-order costs. No ordering cost is calculated in the beer game whose optimal solution results from a base-stock policy. The game was initially introduced by a group of faculty members in Sloan School Management at Massachusetts Institute of Technology in order to show the difficulty with managing dynamic systems. This game is a sample of a dynamic system in supply chain which delivers beer from a beer producer to the end customer. Although supply chain structure and rules of playing the game are very simple, the complex behavior of this dynamic system is interesting. The game is categorized in a group of games illustrating bullwhip effect (Devika et al. 2016, Croson and Donohue 2006). This effect happens unintentionally whenever seeking minimum cost. It happens when the order variation in upstream moving node increases in the network. Lee et al. (1997) and Sterman (1989) explained some rational and behavioral causes of the occurrence of the aforementioned effect, respectively.

There is no algorithm to find optimal base-stock levels whenever a stock-out is observed in a non-terminal agent. Sterman (1989) analyzed the dynamic of environment by considering the dynamic of stock system and the model of environment flows in the beer game. One of the main points of the game is that no data sharing, which can be inventory value or cost amount, happens until the end of the game. Therefore, each agent has a partial information about environment, which leads to observing a *POMDP* model. The cost function used in his work was the summation of linear holding and the stock-out (backorder) cost whose coefficients are 0.5 and 1, respectively. This ratio is used in Case Study 2 of the present research.

Giannoccaro and Pontrandolfo (2002) presented a method to find the best decisions about inventory management containing Markov Decision Processes (*MDP*) and an *AI* method (*RL* approach) to solve *MDP*. Their game consisted of 3 agents whose shipment time and lead-time were stochastic. The *RL* approach was applied in order to find a near-optimal inventory policy based on maximizing the average reward. The reason for applying *RL* was due to its stochastic property as well as its efficiency in large-scale networks. Giannoccaro and Pontrandolfo's RL methods contained three agents whose inventory levels were state variables discretized into 10 intervals and the action number could be between one and thirty. Their methods needed to discretize the inventory level into ten intervals, while it was not generally possible to find an appropriate division of time intervals. This is a defect, which is overcome in this research. Kimbrough et al. (2002) recommended an agent-based approach for serial multi-agent so as to track demand, delete the Bullwhip effect, discover the optimal policies which were known, and find efficient policies under complex scenarios where analytical solutions were not known. Their method was a Genetic Algorithm along with a Joint Action Learners (*JAL*s). They used "$x + y$" rule, in which "x" refers to the amount of demand or order and based on this amount, order quantity equals "$x + y$". By applying this rule, track demand was carried out and the bullwhip influence was eliminated. This resulted in discovering the optimal policies when these policies could be found. In order to determine the order quantity, Chaharsooghi et al. (2008) proposed an approach similar to the method of Kimbrough et al. (2002) containing two differences. First, they worked with four agents, and second, each game had a fixed length equal to 35 time periods and their state variable consisted of four inventory positions which were divided into nine different intervals. The inventory levels and time intervals were restricted to 4 and 35, respectively, which was a limitation to generalize the work. This problem is resolved in the present research.

Claus and Boutilier (1998) utilized (a simple form of) Q-learning to solve cooperative multi-agent environments. The effect of different features on the interaction between equilibrium selection learning techniques and RL techniques was investigated. They mentioned that Independent Learners (*IL*s) and Joint Action Learners (*JAL*s) were two different types of Multi-Agent Reinforcement Learning (*MARL*). A classic type of Q-learning ignoring the other agents was applied in ILs. On the other hand, *JAL*s learned their action value of related agents by combining RL methods with equilibrium learning methods. Parashkevov (2007) evaluated *JAL*s in stochastic competitive games. His approach was able to obtain the safety value of the game and adapt to changes in the environment.

There are two different solutions for the beer game when special conditions arise. In case of availability of stock-out cost only at the final agent (retailer), Clark and Scarf (1960) presented an algorithm to find the optimum policy for the game as the first solution. In order to determine the optimal policy, Chen and Zheng (1994) and Gallego and Zipkin (1999) suggested a similar approach based on the division of serial network into several single-stage nodes. They defined a convex optimization problem with just one variable at each of these stages. Their method suffered from large-volume calculations of numerical integration as well as huge cost of implementation. Later, Shang and Song (2003) proposed an effective approach based on heuristic methods. The solution of Clark and Scarf (1960) and their followers need to consider the specific data distribution, while there is no need to know the data distribution in the present work.

A stochastic process with fixed joint probability distribution is called a stationary environment. If there is no ordering cost and the environment is stationary, the optimal policy of the beer game is base stock. As Gallego and Zipkin (1999) defined, in this policy, the ordering amount was equal to the difference between a fixed number and the current inventory position. Clark and Scarf (1960) called this constant number a base-stock level and there was no general solution to find the optimal value of the base-stock level when there existed a stock-out cost in any agent except for the final agents (retailers). Gallego and Zipkin (1999), as well as Cheng and Zheng (1994), found optimal solutions by neglecting the stock-out cost. Accordingly, the review of the literature signifies that no definite algorithm was presented when general stock-out was available.

Sterman (1989) presented some relations in order to find the order amount by considering order backlog, in and out shipment flow, on-hand inventory, and expected demand, known as the second solution. He modeled the reactions to shortage or extra inventory value of a four-part serial inventory network. Then, Croson and Donohue (2006) studied the behavioural causes of the bullwhip effect and the subsequent behaviour of the beer game. Recently, Edali and Yasarcan (2014) provided a mathematical model for the game.

Classical supervised *ML* algorithms such as support vector machine, random forest, or supervised *DNN* are inapplicable in this research because of none-availability of historical pairs of input/output data. On the other hand, the present research is designed based on *DQN*. Although this research study implements the *DQN* method into a single-agent model, it can be designed for multi-agent inventory whose agents are *JAL*s and *POMDP*. The agents roughly work similar to the beer game. One difference is that the parameters such as inventory level are unlimited and there is no restriction on their values. Another difference is that the ordering cost is considered by adding the cost per order to the cost function. To the best of the author’s knowledge, there are limitations on the values of some parameters such as inventory level as well as ignoring ordering cost in most of the past *RL* approaches. Also, there is a number of research works to solve the one-agent *MDP* whose parameter values are limited such as inventory capacity. As far as the literature reveals, this is the first work considering the holding, backorder, and ordering costs without any limitation on the values of parameters such as inventory level and without any need to know the demand distribution. Also, in the previous *MDP* studies, lead-time and its related parameters such as on-order inventory or inventory transit are ignored, while they are considered in the present research study.

CHAPTER 3 THEORY AND FORMULATION

In this chapter, some techniques including *RL*, *DP*, and *DQN* are described and compared. The formulation and details of different techniques including *DQN* are studied.

3.1 Reinforcement Learning

One of the best methods to deal with complicated decision making issues is Reinforcement Learning (*RL*) (Sutton and Barto 1998). *RL* is part of Machine Learning (*ML*) acting with agents whose next status is influenced by action selection. This selection is examined in order to maximize/minimize the future reward/cost by interaction of the agent and the environment.

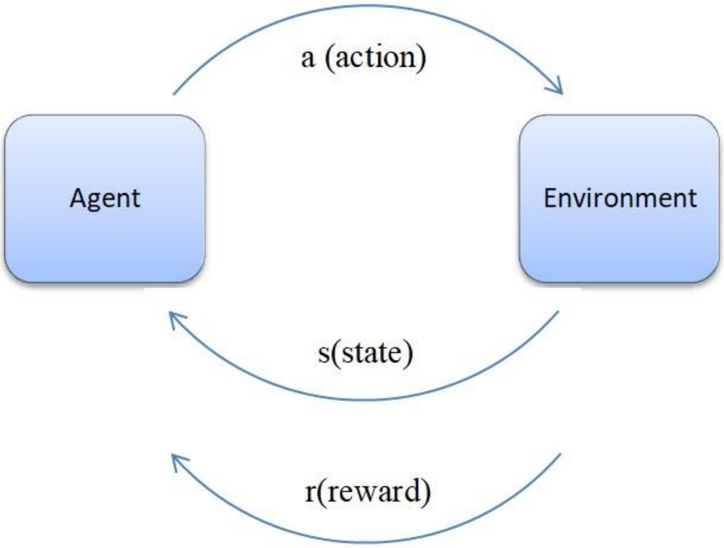

Figure 3.1 Interaction of agent with environment

3.2 Markov Decision Process

A Markov Decision Process (*MDP*) is defined as $M = (S, A, P, R)$, where S is a set of states, A is a set of actions, $P(S \times S \times A \rightarrow [0,1])$ is transition probability distribution, and $R(S \rightarrow R)$ is reward. To be more precise, in each time step t, a state is the situation of agent and action a_t is a command in order to reach next state s_{t+1} from the current state s_t by following the state policy $\mathrm{\pi(s)}$. Generally, policy π is a behaviour function choosing actions given states $(a = \mathrm{\pi(s)})$ and the transition probability, $\mathrm{P}(s_{t+1} = \mathrm{s}' | s_t = \mathrm{s}, \mathrm{\pi(s)} = a)$, shows the probability of transition from state s to state s' by taking action a. The general goal of RL is to maximize the expected discounted sum of the rewards over running on an infinite time horizon (Eq. (3-1)).

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{3-1}$$

where γ is the discount factor. However, in case of large state-space and long-running time of *RL* approach, P and R are very large and are not previously known while the system is a *MDP* observing the states and rewards after taking an action. The value of a policy is determined by solving a linear system or by doing an iteration which is similar to value iteration. Finding the optimal policy with unknown $P$and $R$is a challenging task.

3.3 Comparison of Different Techniques

In order to better understand different approaches, comparisons have been made in this section.

3.3.1 Reinforcement Learning versus Supervised Learning

Supervised Learning is able to solve many problems containing image classification and text translation. However, supervised learning is unable to play a game efficiently. For instance, a dataset containing the history of all the cases of “Alpha-Go” game played by humans could potentially use the state as input x and the optimal decisions taken for that state as output labels y. Although it would be a nice idea in theory, in practice some drawbacks exist as follows:

1. The above-mentioned data sets do not exist for the entire domain.

2. It might be expensive and unfeasible to create the above-mentioned data sets.

3. The method learns to imitate a human expert instead of really learning the best possible policy.

RL wants to learn actions by trial and error. The objective function of a RL algorithm, $E(\sum rewards)$, is an expectation of a system which is unknown. In contrast, supervised learning algorithms tend to find $min_{\theta} loss\ (x_{train}, y_{train})$, in which θ shows the parameters of the algorithm and (x_{train}, y_{train}) are pairs of training set. The supervised learning algorithms learn the optimal strategy by sampling actions and then observing which one of the actions leads to the target output. Contrary to the supervised approach, learning the optimal action in *RL* approach is not conducted based on one label, rather based on some time-delayed labels called rewards, which then determine the performance of the action. Therefore, the goal of *RL* is to take actions in order to maximize reward**.**

A *RL* problem is described as a Markov decision process which is memory less so that every parameter should be known from the current state. Supervised learning learns by examples of pairs of desired inputs and outputs, while RL learns by agents and guesses the correct output. *RL* receives some feedback from the quality of its guess, whereas it does not mention whether this output is the correct one and there is probably some delay in seeing the feedback. *RL* learns either by exploration or by trial and error. The three basic problems in the area of *RL* are the curse of dimensionality, learning from interaction, and learning with delayed-consequence.

3.3.2 Reinforcement Learning versus Dynamic Programming

Dynamic Programming (*DP*) is not the same as value or policy iteration conceptually. This is because the DP approaches are the planning methods, which means that they are able to calculate the value function and an optimal policy iteratively by the given transition and a reward function. Dynamic programming is a series of algorithms that can be utilized to calculate optimal policies if the whole model of environment is available as a Markov Decision Process (*MDP*).

Although classical *DP* algorithms are less beneficial in *RL* due to assume a complete model and to be computationally expensive, they are still important from a theoretical standpoint. *DP* needs a full description of the *MDP*, with known transition probabilities and reward distributions that are used by a *DP* algorithm. This property makes it model-based. *DP* is one part of *RL* which is a value-based, model-based, bootstrapping and off-policy algorithm. In summary, *DP* is a planning method, which means that a value function and optimal policy is computed by giving a transition and calculating a reward. On the other hand, Q-learning, which is a special case of value iteration, belongs to a model-free class of *RL* methods due to not utilizing any environmental model. However, model-based methods work based on learning a model, while contrary to the model-free approaches, the samples are kept even after value estimation. The *RL* methods try to reconstruct the transition and reward in order to have better efficiency. A combination of model-free and planning algorithms is presented in model-based algorithms in which fewer sampling is required in comparison with model-free algorithms such as Q-learning. Also, the model-based *RL* algorithms do not need a model similar to *DP* approaches such as value or policy iteration. Therefore, fewer sampling and independence from *DP* modeling are advantages of model-based *RL* algorithms in comparison with model-free and classical dynamic programming approach, respectively.

3.3.3 Q-learning versus DQN

Q-Learning is one of the pioneer *RL* approaches presented by Watkins (Watkins and Dayan 1992) and is applied as a baseline of *RL* results. Although the Q-Learning approach is a powerful algorithm, it is not applicable in all cases. This is because it requires to know all pairs of states and actions while is generally impossible. Therefore, to tackle this problem, an approximation of Q-function can be found by a *NN* and if *NN* is replaced by *DNN* as an action approximator, the algorithm is *DQN*. This algorithm was introduced by the DeepMind company in 2013 and states and Q-value of the actions were its inputs and outputs, respectively.

The general formula for a Q-function is given as:

$$\mathrm{Q}(s,\mathrm{a}) = \mathrm{r} + \gamma\ \mathrm{max}_{\mathrm{a}'}(\mathrm{Q}(\mathrm{s}',\mathrm{a}')) \tag{3-2}$$

and the general formula for *DQN* is

$$\mathrm{Q}(\mathrm{s},\mathrm{a};\Theta) = \mathrm{r} + \gamma\ \mathrm{max}_{\mathrm{a}'}(\mathrm{Q}(\mathrm{s}',\mathrm{a}';\overline{\Theta})) \tag{3-3}$$

In the above formulae, r and γ are reward and discount factor, s, s', a, θ and $\bar{\theta}$ are state, next state, action, parameters of *NN*, and parameters to compute the target of *NN*, respectively.

On the other hand, in *DQN*, a neural network is added to a very large Q-table in which there is a large number of states and actions. The neural network is applied in order to compress the Q-table by setting the parameters of neural networks. Also, since the number of NN nodes is supposed to be constant, these parameters are restricted to coefficient weights of neural network. By smart tuning the configuration parameters of the structure explained, an optimal Q-function can be found by various neural network training algorithms. If f_θ is a neural network with weight parameters θ and input s, the Q-function can be written as $Q(s,a) = f_\theta(s)$.

A Q-learning environment contains reward and observation and gives them to an agent in order to decide an action. In *DQN*, the agent is replaced with a function showing the weights of a *DNN*. A *DRL* approach learns a parameterized function f_θ; its loss function is differentiable with respect to θ and optimization is performed with gradient-based algorithms. Also, a difference between the Q-learning and *DQN* is presented in Figure 3.2. θ is a set of features of neural networks (if the number of nodes and layers in general structure of *DNN* are considered constant, θ is considered coefficient weights). Also, s and a show state and action, respectively.

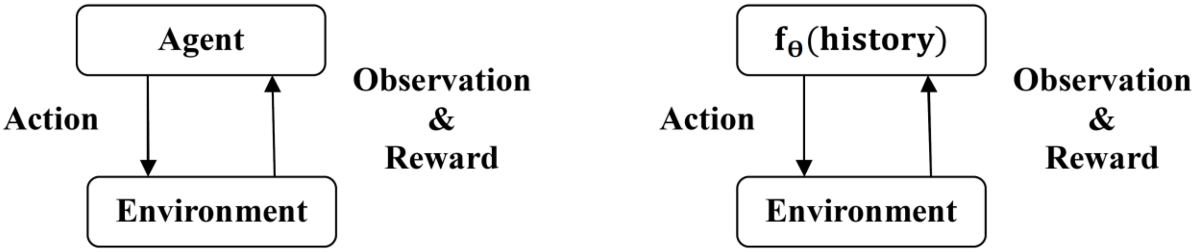

Figure 3.2 Q-learning (Left) versus *DQN* (Right)

3.4 Different Types of Reinforcement Learning Algorithms

Model-based and model-free are two different types of *RL* techniques. The model-based agent builds a transition model of the environment and plans (e.g. by lookahead) using the model. In the model-based algorithms, if there are sufficient samples of each state parameter, the estimations of reward and transition probability converge to the correct *MDP*, value function and policy. However, obtaining a sufficient number of samples is still a challenge to be solved. A drawback of the model-based method is that the actual *MDP* model should be made when the size of state is too large. In addition, a policy-based *RL* approach searches directly for the optimal policy π^* which is the policy achieving maximum future reward. Also, value-based *RL* approach estimates the optimal value function $Q^*(s,a)$, which is the maximum value achievable under any policy.

The agents of the model-free algorithms such as Q-learning and policy gradient can learn action and policy directly. In addition, a policy-based reinforcement learning approach searches directly for the optimal policy π^* which is the policy achieving maximum future reward. Also, value-based *RL* approach estimates the optimal value function $Q^*(\mathrm{s},\mathrm{a})$ which is the maximum value achievable under any policy. Temporal Difference (*TD*), State-Action-Reward-State-Action (*SARSA*), and Q-learning are some examples of model-free *RL* algorithms working based on temporal difference. The main benefit of model-free RL approaches is the application of function approximation in order to represent the value function without having to derive. If function approximation with parameters θ is expressed as $f_\theta(s)$, TD update is $\theta \leftarrow \theta + \alpha(r + \gamma f_\theta(s') - f_\theta(s))\nabla_\theta f_\theta(s)$, where s' is the next state, $\nabla_\theta f_\theta(s)$ is the gradient of $f_\theta(s)$, α is learning rate and γ is discount factor. This process is similar in *SARSA* and Q-learning.

3.4.1 Q-learning

Q-learning is a model-free approach learning task that applies samples from the environment. It is also an off-policy algorithm due to learning with a greedy strategy $a = max_a\, Q(s,a)$ and it guarantees sufficient exploration of states due to following a behaviour distribution. This behaviour distribution is chosen by using a ϵ- greedy algorithm, which will be explained in the subsequent sections. Q-function is the main part of Q-learning. $Q(s,a)$ determines the maximum discounted future reward by performing action a when the current state is s. It also estimates the selection of action a in state s. However, “Why is Q-function useful?” and “How is Q-function obtained?” are two main questions worth answering. To achieve this, it is better to see the structure of Q-function. If a strategy to win a complex game is unknown, the players cannot play well. However, the situation is different when a guide book containing hints or solutions is available. The Q-function is similar to this guidebook. If a player is in state s and there is a need for action selection, the player selects the action obtaining the highest Q-value. $\pi(s)$ is the action associated with state s under policy π given as:

$$\pi(s) = \mathrm{argmax_a}\ \mathrm{Q(s,a)} \qquad (3\text{-}4)$$

Total future reward is R_t written as:

$$\mathrm{R_t} = \sum_{\mathrm{i=t}}^{\mathrm{n}} \mathrm{r_i} \qquad (3\text{-}5)$$

in which r_i is the reward for each state.

Since the environment is stochastic, there is uncertainty about future increases during running time steps. As a result, calculation of R_t is not possible, and consequently, discounted future reward is calculated instead of R_t as follows:

$$\mathrm{R_t = r_t + \gamma r_t + \cdots + \gamma^{n-t} r_n} \qquad (3\text{-}6)$$

As mentioned previously, the Q-function is the maximum discounted future reward in state s and action a expressed below:

$$\mathrm{Q(s_t, a_t) = \max R_{t+1}} \qquad (3\text{-}7)$$

Therefore, the Q-function can be expressed as the summation of reward r and maximum future reward for next state s' and action a' as follows:

$$\mathrm{Q(s, a) = r + \ \gamma * \ \max_{a'} Q(s', a')} \qquad (3\text{-}8)$$

This equation is known as Bellman equation. Q-function is solved with an iterative method using an experience (s, a, r, s'). Considering $r + \gamma * max_{a'} Q(s', a')$ as an estimator and $r + \gamma * max_a Q(s, a)$ as a predictor, making a Q-table similar to performing a regression. The loss function of Q-learning is a Mean Squared Error (*MSE*) given by:

$$\mathcal{L} = [\mathrm{r + \ \gamma * \ \max_{a'} Q(s', a') - Q(s, a)}]^2 \qquad (3\text{-}9)$$
$$\leftarrow - - \ \text{target} \ - - \rightarrow$$
$$\leftarrow - - - - \ - - \text{TD error} - - - - \rightarrow$$

Optimization of Q-function with an experience (s, a, r, s') is performed by considering the smallest *MSE* as loss function. If $\mathcal{L}$ tends to decrease, the convergence of Q-function to optimal value occurs.

3.4.2 From RL to DQN

The *RL* techniques are divided into two categories: Tabular Solution Methods and Approximate Solution Methods (Sutton and Barto 1998; Sutton and Barto 2018). If the probability and the reward of transition from state s to state s' by taking action a are given, optimal policy could be found by linear programming or by a type of dynamic programming method such as value iteration or policy iteration. In most cases, the process is not completely Markov Decision Process (*MDP*), meaning that the history is somehow important, and as a result, a Semi Markov Decision Process (*SMDP*) exists. This means that in a system with reasonable running time, in the cases of large state-space and large action-space, finding the optimal policy to solve the *MDP*s is not possible due to curse of dimensionality. In contrast, in the cases with a large number of states or action spaces, observing full state spaces is not possible for decision makers (agents). This leads to partial observability of state variables called Partial Observable *MDP* (*POMDP*). Since it is hard to determine the appropriate Q-values in a *POMDP*, the approximation of Q-values is made in the Q-learning algorithm (Sutton and Barto 2018). To end this, first, linear regression was used as a function approximator (Melo and Ribeiro 2007), which was replaced by a non-linear function approximator such as neural network due to its ability to find more reliable accuracy.

To utilize function approximation, it was necessary to extract a number of features until the early 2010's. For instance, object recognition methods employed hand-made features and linear classifier learners (Patel and Tandel 2016). However, from 2012, most of vision techniques started utilizing *DNN* for feature extraction and going towards end-to-end whole pipeline optimization (Szegedy et al. 2013). *DL* is very successful in learning when the features are unknown. As a result, a combination of *RL* and *DL* called *DRL* has received much attention recently (Li 2017). Mnih et al. (2013) proposed an algorithm for *DRL* called *DQN* in 2013. Since 2013, many researchers have worked on this issue and the algorithm is ameliorated and completed significantly (Li 2017). However, the algorithm was not widely used by researchers until the DeepMind group released more details of their approach in 2015 (Mnih et al. 2015). This is because they encountered some difficulties such as observing unstable or even divergent Q-value as Q-function approximator resulting from non-stationary and correlations in the sequence of the observations so as to implement neural network (Mnih et al. 2013). To overcome the challenge, they used the Experience Replay (*ER*) first introduced by Watkin and Dayan (1992). Schaul et al. (2015) ameliorated their previous research work (Mnih et al. 2015) using the prioritized *ER* technique. Traffic light control in vehicular networks is its application in transportation (Liang et al. 2018).

3.4.3 DQN

DQN is a combination of Q-learning and Neural Network (*NN*), in which the function approximation of Q-learning is a *DNN*. *DQN* is a Q-learning approach whose action is chosen based on a *DNN*. Actions are related to the outputs of *NN*, whereas states of the *RL* are the inputs of NN. Also, *DQN* learns a Q-function by minimizing Temporal Difference (*TD*) errors. A transition (s, a, r, s') is observed and *TD* error tends to make $Q(s, a)$ as close as possible to $r + \gamma max_{a'} Q(s', a')$. Action can be selected arbitrarily in off-policy algorithms with a ϵ-greedy policy based on the current Q-value. To be more precise, value function $Q^{\pi}(s, a)$ is the expected total reward from state s and action a under policy π which can be unrolled recursively as follows:

$$Q^{\pi}(s, a) = \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots |s, a] = \mathbb{E}_{s'}[\, \mathrm{r} + \gamma\, Q^{\pi}(\mathrm{s}', \mathrm{a}')|s, a] \qquad (3\text{-}10)$$

Also, optimal value function $Q^*(s, a)$ can be unrolled recursively as:

$$Q^*(s, a) = \mathbb{E}_{s'}[\, \mathrm{r} + \gamma\, max_{a'}\, Q^*(\mathrm{s}', \mathrm{a}')|s, a] \qquad (3\text{-}11)$$

where value iteration algorithms solve the Bellman equation as follows:

$$Q_{i+1}(s, a) = \mathbb{E}_{s'}[\, \mathrm{r} + \gamma\, max_{a'}\, Q_i(\mathrm{s}', \mathrm{a}')|s, a] \qquad (3\text{-}12)$$

The value function represented by deep Q-network whose parameters are θ is given by:

$$Q(s, a, \theta) \approx\; Q^{\pi}(s, a) \qquad (3\text{-}13)$$

The objective function defined by mean-squared error in Q-values is expressed as:

$$\begin{aligned} \mathcal{L} &= \mathbb{E}[(\mathrm{r} +\; \gamma *\; \max_{\mathrm{a}'} \mathrm{Q}(\mathrm{s}', \mathrm{a}', \bar{\theta}) - \mathrm{Q}(\mathrm{s}, \mathrm{a}, \theta))^2] \qquad (3\text{-}14) \\ &\quad \leftarrow - - \text{target} - - \rightarrow \\ &\quad \leftarrow - - - -\; - - - \text{TD error} - - - - - \rightarrow \end{aligned}$$

which leads to the following gradient function:

$$\frac{\partial \mathcal{L}(\theta)}{\partial \theta} = \mathbb{E}[(\mathrm{r} +\; \gamma *\; \max_{\mathrm{a}'} \mathrm{Q}(\mathrm{s}', \mathrm{a}', \bar{\theta}) - \mathrm{Q}(\mathrm{s}, \mathrm{a}, \theta)] \frac{\partial \mathrm{Q}(\mathrm{s}, \mathrm{a}, \theta)}{\partial \theta} \qquad (3\text{-}15)$$

As a result, the end-to-end *RL* is optimized by an optimizer using $\frac{\partial \mathcal{L}(\theta)}{\partial \theta}$.

3.4.4. Optimizers

There are many optimizer techniques among which Stochastic Gradient Gdescent (*SGD*) and ADAptive Moment estimator (*ADAM*) are more favorite. The batch methods utilize the entire training sets in order to update the parameters in any iteration with a tendency to converge to local optimal. For a large dataset, the speed of finding the cost and gradient of the full training data set is very low. Also, a batch optimization approach is not a suitable method to merge new data in the online settings. In order to resolve these problems, *SGD* approaches follow the negative gradient of objective after a few training samples. Since the cost of the running backpropagation over the entire training set is high, it is helpful to use *SGD* in neural network setting. In *SGD*, the parameters θ of objective $J(\theta)$ are updated with $\theta{=}\theta-\alpha\nabla\theta E[J(\theta)]$, where $\nabla\theta$ is the gradient of θ and α is the learning rate. If *SGD* uses a few training samples, it easily disappears with the update expectation and gradient computation. As a result, the update is given by a new formula extracting $\left(x(i), y(i)\right)$, where $x(i)$ and $y(i)$ are the i^{th} pair of training set, from the training data as follows:

$$\Theta{=}\Theta-\alpha\nabla\Theta\mathrm{J}(\Theta;\mathrm{x(i)},\mathrm{y(i)}) \qquad (3\text{-}16)$$

Updating the parameters in *SGD* is based on a few trainings or mini-batch samples. This is due to variance reduction in the updated parameters, leading to a more stable convergence. Also, it can benefit from the optimized matrix operations used in computation of cost and gradient. The learning rate of stochastic gradient descent, α, is lower than that of batch gradient descent due to the existence of more updating variance. The decisions are made to find the correct learning rate and time of updating the learning value.

Also, Adaptive Moment Estimation (*Adam*) computes the adaptive learning rates of each parameter which not only stores an exponentially decaying average of past squared gradient v_t, but also keeps an exponentially decaying average of past gradients m_t which is similar to momentum. Adam behaviour is similar to heavy ball with friction which prefers to flat minima in the error surface, whereas momentum pushes a ball running down a slope. m_t and v_t estimate the first (mean) and the second (the uncentered variance) momentum of the gradients, respectively. They are expressed by:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \qquad (3\text{-}17)$$

$$v_t = \beta_2 v_t + (1-\beta_2) {g_t}^2 \qquad (3\text{-}18)$$

The initialization of m_t and v_t are zero vectors, while during the initial time steps and especially with a small decay rates (β_1 and β_2 are close to 1), there are biases towards zero. To counteract this problem, corrections for the first and the second moments of bias are written as follows:

$$\widehat{m_t} = \frac{m_t}{1-\beta_t^1} \qquad (3\text{-}19)$$

$$\widehat{v_t} = \frac{v_t}{1-\beta_t^2} \qquad (3\text{-}20)$$

Using the above equations for updating parameters (θ_{t+1}), the Adam update rule is given as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\epsilon + \sqrt{\widehat{v_t}}} \widehat{m_t} \qquad (3\text{-}21)$$

Also, *SGD* examines the error calculation and updates the reward. The general formula for the Q-function is expressed as follows:

$$\mathrm{Q(s,a) = r + \gamma \max_{a'}(Q(s',a'))} \qquad (3\text{-}22)$$

In Mean Square Error (*MSE*), Temporal Difference error (*TD*-error) and target are calculated as follows:

$$\mathrm{Q_\pi(s,a) \leftarrow\ Q_\pi(s,a)\ + \gamma\ [(r + \max_{a'} Q_\pi(s',a') - Q_\pi(s,a)]} \qquad (3\text{-}23)$$

$$\leftarrow - - \text{target} - - \rightarrow$$

$$\leftarrow - - \;\; - - \text{TD error} - - - - \rightarrow$$

in which s' is s_{t+1} and a' is a_{t+1}.

It should be mentioned that in the aforementioned general formulae, maximization of the reward is the goal of the algorithm. However, in this research, the objective is to minimize the average reward cost. *TD* error calculates the difference between expectation of Q-approximation in the future plus the reward and its present value as evaluated by the neural network. If the state is the terminal one, then reward at step j, y_j, is given as:

$$y_j \leftarrow r_j \tag{3-24}$$

Otherwise,

$$y_j \leftarrow r_j + \gamma. \min_a Q\left(s_j, a_j, \overline{\Theta}\right) \tag{3-25}$$

Since there is no definite terminal state in this problem, running for a definite number of time steps is considered as the terminal state. Although a linear function approximation is mostly utilized in *RL* approaches, a nonlinear function approximation is sometimes used. This nonlinear function can be found by a Neural Network (NN). Mnih et al. (2013) employed a neural network function approximator with parameters Θ as a Q-network. Training the Q-network can be carried out by minimizing a sequence of loss functions $\mathcal{L}_i(\Theta_i)$ which change in each iteration. Also, y_i is the goal of iteration i and $\rho(s, a)$ is defined as a probability distribution over sequences s and action a. The parameters of the previous iteration Θ_{i-1} are constants during the optimization of the loss function $\mathcal{L}_i(\Theta_i)$ expressed as:

$$\mathcal{L}_i(\Theta_i) = \mathbb{E}_{s,a\sim\rho(.)}[(y_j - Q(s_j, a_j, \Theta))^2] \tag{3-26}$$

where y_j is the target of iteration i, which is written as:

$$y_i = \mathbb{E}_{s'\sim\epsilon}[r + \gamma \max_{a'} Q(s', a'; \Theta_{i-1})|s, a] \tag{3-27}$$

Another important point is that the targets are dependent on the network weights. In contrast, in supervised learning approaches, the weights are considered as constants before starting to perform learning. To find Mean Square Error (*MSE*), a gradient descent is conducted on $(y_j - Q(s_j, a_j, \Theta))^2$ and since *MSE* is differentiable, the derivative of the loss function with respect to the weights is calculated as follows:

$$\nabla_{\Theta_i}\mathcal{L}_i(\Theta_i) = \mathbb{E}_{s,a\sim\rho(.);s'\sim\epsilon}[(r + \gamma \min_{a'} Q(s', a'; \Theta_{i-1}) - Q(s, a; \Theta_i))\ \nabla_{\Theta_i} Q(s, a, \Theta_i)] \tag{3-28}$$

It is suitable to optimize the loss function by stochastic gradient descent, instead of calculating the expectation in the above formula. However, if weight updating is conducted in each time step and the expectations are replaced by a single sample from the behaviour distribution ρ and the environment ϵ, then a familiar Q-learning algorithm is reached (Mnih et al. 2015).

3.5 Exploration versus Exploitation (ϵ-greedy algorithms)

One of main challenges in machine learning algorithms is “exploration versus exploitation”. It is similar to real life when a person decides to follow an existing policy or to check out a new policy. To obtain the maximum future reward, agents need to find a balance between exploitation (greedy) and exploration (ϵ-greedy algorithms). If the dynamic of a system is unknown, exploring actions or exploiting the current knowledge gives the best answer. When a bad initial state-action is chosen, the algorithm gets stuck in local minimum and can never explore further. To resolve this problem, instead of selecting the action based on greedy algorithm, a policy is explored until a good estimation of value function is found.

3.6 Improvement of DQN

The *RL* approaches suffer from instability or even divergence when active-values (Q-function) is represented by a nonlinear function approximator such as a neural network (Tsitsiklis and Van Roy 1997). To reach a more stable learning algorithm, *DQN* can be improved by different techniques including experience replay, target network and skipping frames, leading to more stable results.

3.6.1 Experience Replay

Lin (1993) applied a mixture of RL and neural network for robots. He integrated back propagation and temporal difference. Over-fitting in *DNN* happens occasionally and easily. As a result, it is hard to produce various experiences. In order to tackle this problem, Experience Replay *(ER)* memory stores all important data parameters including reward, action, state, and next state. Updating the neural network is carried out by making mini-batches. However, the *ER* technique is a simple, effective technique that resolves the temporal credit assignment problem. This technique reduces the correlations among the training data of updating *DNN* and decreases the variance of the output. Another benefit is that mini-batches increase the learning speed that is effective in decreasing time, which is essential to learn huge data. Also, *ER* reuses transitions from historical data which avoids catastrophic forgetting. A general structure of *ER* is displayed in Figure 3.3. In order to provide data-set from the experience of the agent, in each time step t, action a_t is taken according to ϵ-greedy and then, transition, $(s_t, a_t, r_{t+1}, s_{t+1}) = (s, a, r, s')$, is stored in replay memory D. Finally, Mean Square Error (*MSE*) between Q-network and Q-learning targets is optimized, e.g, $\mathcal{L}(\Theta) = \mathbb{E}_{s,a,s',a' \sim D}[(\mathrm{r} + \gamma \mathrm{min}_{\mathrm{a}'} \mathrm{Q}(\mathrm{s}', \mathrm{a}'; \Theta) - \mathrm{Q}(\mathrm{s}, \mathrm{a}; \Theta))^2]$

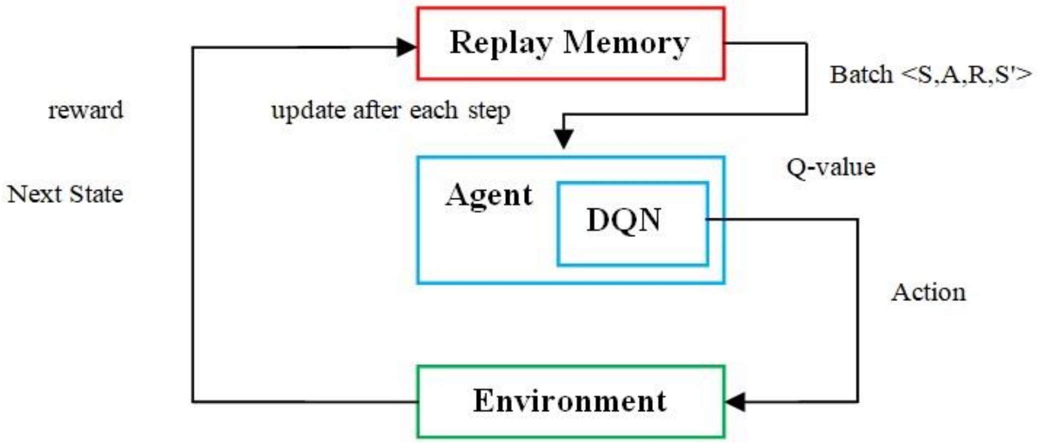

Figure 3.3 Experience Replay (*ER*) in DQN

3.6.2 Target Network

Target function changes frequently with *DNN* during the calculation of Temporal Difference (*TD*) error leads to instability and oscillation. This instability makes the training of *DNN* more complicated. It is interesting that targets is related to the network weights which is in contrast with the targets used for supervised learning, which are constant before learning starts. To get rid of this complexity, parameters of target function are considered constant in most steps. Target Q-function, $\mathrm{Q}(s', \mathrm{a}')$ is computed w.r.t fixed parameters $\bar{\theta}$ as:

$$\mathrm{r} + \gamma\ \mathrm{max}_{\mathrm{a}'} \mathrm{Q}(s', \mathrm{a}', \bar{\theta}) \qquad (3\text{-}29)$$

where α and γ are learning and discount rate, respectively. Also, $Q(s', a')$ shows the Q-function of next state and action. Also, *MSE* optimizes between Q-network and Q-learning targets as follows:

$$\mathcal{L}(\theta) = \mathbb{E}_{\mathrm{s,a},s\prime,\mathrm{a}\prime\sim D}\left[\left(\mathrm{r} + \gamma\mathrm{min}_{\mathrm{a}\prime}\mathrm{Q}(\mathrm{s}', \mathrm{a}'; \bar{\theta}) - \mathrm{Q}(\mathrm{s}, \mathrm{a}; \theta)\right)^2\right] \quad (3\text{-}30)$$

while the parameters of target function are replaced with those of the last neural network every several hundreds or even thousands steps in target network techniques, i.e. $\bar{\theta} \leftarrow \theta$.

3.6.3 Skipping Frames

Lead-time in a serial network causes delays in observing the influence of action selection on reward function. Also, the amount of reward function may be related to the previous multiple periods. Therefore, skipping frames is a technique which calculates Q-value every k frames and Q-value considers the last k frames as inputs. This technique reduces the computational cost and gathers more experiences. k should be a reasonable number showing the minimum time required for making a demand to be met.

3.6.4 Different Loss Function

Mean Square Error (*MSE*) of Q-function pays more attention to large errors in comparison with Mean Absolute Error (*MAE*). In contrast, *MAE* treats large and low errors similarly. The intuition behind *MSE* is that it is better to have a larger priority in order to minimize large errors rather than small ones.

CHAPTER 4 INVENTORY CONTROL SOLUTION

In order to solve the inventory control problem, an approach based on a combination of DL and RL is presented in this chapter. It should be mentioned that since all the formulation in this research is planned for multi-agent (multi-stage) problems, all the experiments are carried out for only single stage (one agent) whose real input and output are demand and ordering value, respectively. The minimization of cumulative long-run system cost whose components are holding, backorder, and ordering costs is desired. This selection is made because holding the products as well as making an order impose cost for the inventory manager. Also, backorder determines the inventory shortfall which represents the number of unmet demands waiting to receive inventory. In addition to holding and fixed ordering costs, it is important to track backorder value so as to minimize the total cost. To achieve this, some main features of inventory such as the inventory position, which is the summation of inventory level and inventory transition, should be computed. Accordingly, it is important to find the main features of the agent affecting the inventory level and inventory transition values.

4.1 Main Features of Inventory Control

There is a number of features which are critical in the field of inventory control. Some are related to their past amounts or are interrelated to each other, whereas the others are independent random or deterministic variables. These parameters are listed in Tables 4.1 and 4.2.

4.1.1 Random Features

Demand and lead-time are random features of inventory control as shown in Table 4.1, in which superscript i of D_t^i and LT_t^i is the agent's (stage's) index number. This is done in order to generalize the current single-agent model to multi-agent model. As a result, the agent $i-1$ and $i+1$ are seen in Tables 4.1 and 4.2, showing the interaction of one agent with its superior/prior agent in general. However, the problem is solved for one agent in this research *(i=1)*.

4.1.2 Interrelated Features

Some of the features in inventory control are related to each other. The relations of these parameters given in Table 4.2, will be presented in the next sections. It should be noted that D_t^i is

Table 4.1 Random features

Feature	**Description**
D_t^i	demand received from agent $i-1$ at time t
LT_t^i	lead-time for agent i at time t

Table 4.2 Interrelated features

Feature	**Description**
IL_t^i	inventory level at time t for agent i
IT_t^i	inventory transition at time t for agent i
D_t^i	demand received at time t from agent $i-1$
O_t^i	order sent to agent $i+1$ at time t
OO_t^i	on-order item (ordered item from agent $i+1$ but not received yet) at time t

displayed in both Tables 4-1 and 4-2 because although it is independent of the other features, it changes stochastically and randomly each time, and as a result, its effect on the other features changes over time. In contrast, LT_t^i is considered constant at all times ($LT_t^i = 2$), and therefore, its effect on the other features does not change over time. Thus, it is not seen in the interrelated features. Also, a general list of the different parameters of single-echelon inventory agent for a time step is illustrated in Figure 4.1. If lead-time equals two, on-order consists of part 1 and part 2 displaying the orders sent in the previous time step and the current time step, respectively. All of the other parameters are defined in Table 4.1.

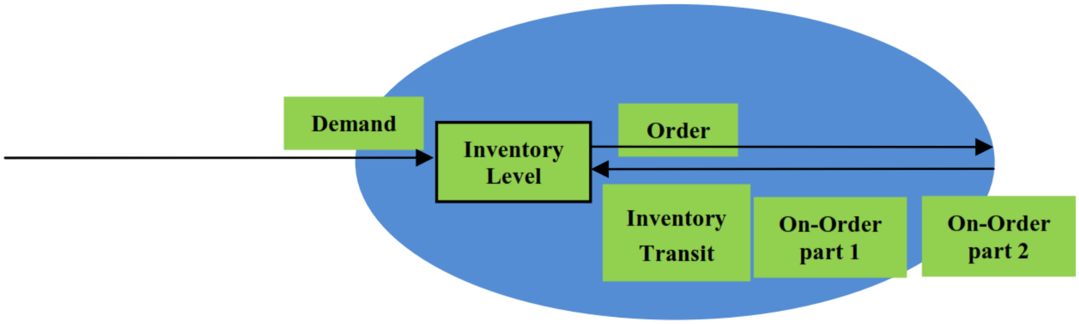

Figure 4.1 A general list of the different parameters of an inventory agent

4.2 Relations among Features

4.2.1 Relations among On-Order, Inventory Transition, and Order

In general, the OO_t is the number of on-order items at time t, which is calculated based on inventory transits as follows:

$$\mathrm{OO_t} = \sum_{\mathrm{v=1}}^{\mathrm{LT}} \mathrm{IT_{t+v}} \tag{4-1}$$

For instance, if lead-time is equal to two, then the on-order value at the current time is the summation of inventory transit at the next two times. This means that all inventories transited at the next LT time steps are added to obtain on-order at the current time while it is sufficient to find inventory transition in order to find on-order inventory. The inventory transition is the ordering value which arrives LT time steps after ordering, expressed in the following relation:

$$\mathrm{IT_{t+LT}} = \mathrm{O_t} \tag{4-2}$$

Therefore, it is important to find a relation between demand and order at each time.

4.2.2 Relations between Demand and Order

Kimbrough et al. (2002) presented a relation between demand and order mentioning that at each time *t* and in each agent i, order O_i^t is the summation of demand D_i^t and a value x_i^t. Also, there is a number of time step delays in observating the rewards. As a result, a memory of states with size k is considered, where k is the number of the recent observations of demands and orders.

4.2.3 Relations among Different Features of DNN

If lead-time *(LT)* is two for all interactions of an agent, the features are updated using Eqs. (4-3) to (4.10). Eq. (4-3) is based on Kimbrough rule. Eq. (4-4) mentions that on-order at each time is the summation of the current order and the previous order, which results from the value of lead-time $(LT = 2)$. Also, since lead-time is two, the inventory transit at the next time is the ordering value at the previous time (Eq. (4-5)) and the change in the inventory level is the summation of inventory transit minus demand (Eq. (4-6)). Eq. (4-7) expresses the relation between order and inventory transit, showing that the inventory transit after passing the lead-time is the current order. Consequently, Eqs. (4-8) to (4-10) indicate that the overall cost is the summation of linear holding cost, linear shortage cost, and fixed ordering cost, in which the linear holding cost is holding coefficient C_h times positive inventory level, the linear shortage cost is shortage coefficient C_p times negative inventory level, and if there is an order, the ordering cost is C_h. D_t^i represents the demand at time t and x_t^i shows the difference between order and demand. Also, D_t^i and x_t^i are extracted from observation and learning with a *DNN*, respectively. Order is assumed to be non-negative.

$$\mathrm{O}^{\mathrm{i}}_{\mathrm{t+1}} = \mathrm{D}^{\mathrm{i}}_{\mathrm{t}} + \mathrm{x}^{\mathrm{i}}_{\mathrm{t}}\,, \mathrm{O}^{\mathrm{i}}_{\mathrm{t+1}} \geq 0 \tag{4-3}$$

$$\mathrm{OO}^{\mathrm{i}}_{\mathrm{t+1}} = \mathrm{O}^{\mathrm{i}}_{\mathrm{t}} + \mathrm{O}^{\mathrm{i}}_{\mathrm{t+1}} \tag{4-4}$$

$$\mathrm{IT}^{\mathrm{i}}_{\mathrm{t+1}} = \mathrm{O}^{\mathrm{i}}_{\mathrm{t-1}} \tag{4-5}$$

$$\mathrm{IL}^{\mathrm{i}}_{\mathrm{t+1}} = \mathrm{IT}^{\mathrm{i}}_{\mathrm{t}} + \mathrm{IL}^{\mathrm{i}}_{\mathrm{t}} - \mathrm{D}^{\mathrm{i}}_{\mathrm{t}} \tag{4-6}$$

$$\mathrm{IT}^{\mathrm{i}}_{\mathrm{t+LT}} = \mathrm{O}^{\mathrm{i}}_{\mathrm{t}} \tag{4-7}$$

$$\mathrm{Cost}^{\mathrm{i}}_{\mathrm{t+1}} = \mathrm{C_h}.\ {\mathrm{IL}^{\mathrm{i}}_{\mathrm{t+1}}}^{+} + \mathrm{C_p}.\ {\mathrm{IL}^{\mathrm{i}}_{\mathrm{t+1}}}^{-}, \qquad \mathrm{Cost}^{\mathrm{i}}_{\mathrm{t+1}} = \mathrm{Cost}^{\mathrm{i}}_{\mathrm{t+1}} + \mathrm{C_o} \quad \text{if} \quad \mathrm{O}^{\mathrm{i}}_{\mathrm{t+1}} > 0 \tag{4-8}$$

$${\mathrm{IL}^{\mathrm{i}}_{\mathrm{t+1}}}^{+} = \mathrm{IL}^{\mathrm{i}}_{\mathrm{t+1}} \quad \text{if} \quad \mathrm{IL}^{\mathrm{i}}_{\mathrm{t+1}} > 0, \qquad {\mathrm{IL}^{\mathrm{i}}_{\mathrm{t+1}}}^{-} = -\mathrm{IL}^{\mathrm{i}}_{\mathrm{t+1}} \quad \text{if} \quad \mathrm{IL}^{\mathrm{i}}_{\mathrm{t+1}} < 0 \tag{4-9}$$

$$\mathrm{Min} \sum \mathrm{Cost}^{\mathrm{i}}_{\mathrm{T}}\,, \qquad if\ \mathrm{Cost}^{\mathrm{i}}_{0} = 0, T = 500, i = 1 \tag{4-10}$$

4.3 State Variables

The environment is non-stationary because data is unpredictable and cannot be forecasted. The total observations for agent i over time t are given as:

$$\mathrm{OB}^{\mathrm{i}}_{\mathrm{t}} = \left[\mathrm{IL}^{\mathrm{i}}_{1}, \mathrm{OO}^{\mathrm{i}}_{1}, \mathrm{D}^{\mathrm{i}}_{1}, \mathrm{IT}^{\mathrm{i}}_{1}, \mathrm{a}^{\mathrm{i}}_{1}, \ldots, \mathrm{IL}^{\mathrm{i}}_{\mathrm{t}}, \mathrm{OO}^{\mathrm{i}}_{\mathrm{t}}, \mathrm{D}^{\mathrm{i}}_{\mathrm{t}}, \mathrm{IT}^{\mathrm{i}}_{\mathrm{t}}, \mathrm{a}^{\mathrm{i}}_{\mathrm{t}}\right] \tag{4-11}$$

Since there is no sharing information except demand/order, the environment is Partial Observable (*PO*). Also, OB_t^i determines states and since its size grows over time, it is difficult for DQN to find OB_t^i. Therefore, it is not logical to consider all of the observations from the starting point. In order to tackle this problem, skipping frames (the last k periods of states) are considered as state variables and the size of input remains fixed (See Section 4.4.1). Also, there are limits in running time, and as a result, the environment is restricted and it is not completely observable, leading the environment to become a Partial Observable Markov Decision Process (*POMDP*).

4.4 Steps of Algorithm

A general structure of *DQN* is displayed in Figure 4.2. A state is a number of features given as inputs of a *DNN* whose parameters are θ. By choosing a policy π_θ based on state s, *DNN* parameters θ and action a, a new action is taken. As a result, the corresponding reward is found. The next state is found after observing the demand from input and updating the other parts of a state. The *DNN* parameters including weighting coefficients, numbers of nodes and layers show the structure of the *DNN*. Since a fixed number of layers and nodes are used after some training, the main parameters of *DNN* can be reduced to the weighting coefficients between the layers. The frequency of updating the weighting coefficients of a network is one of the hyperparameters of the problem. Figure 4.2. shows the general structure of *DQN* when the state is the input of *DNN*, the parameter of *DNN* is θ, an action is taken by policy $\pi_\theta(s, a)$. After calculating reward and observing one parameter from input, the next state is found.

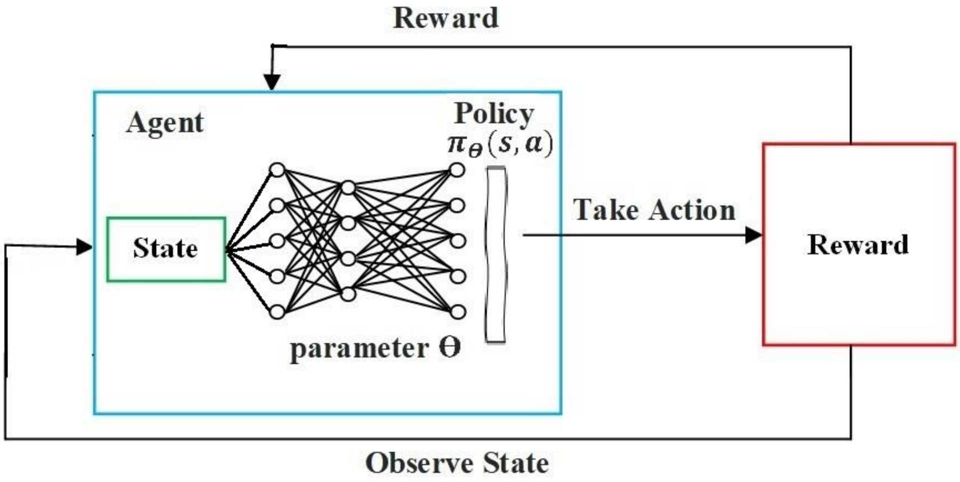

Figure 4.2 A general structure of *DQN*

4.4.1 Implementation of Frame Skipping and ϵ-greedy

The k-frames technique is important to be utilized because in case of availability of lead-time, there is some latency in the environment even in sending or receiving. The size of the frame should be greater than the summation of latency to send an order and receive it from the environment. Therefore, a frame memory whose size is greater than or equal to this summation should be defined as the input (Figure 4.4). In order to consider a memory of size k, small changes in the definition of the current and next states of one agent are applied as follows:

$$\mathrm{S}_{\mathrm{t+1}} = [\mathrm{IT}_{\mathrm{tt+1}}, \mathrm{OO}_{\mathrm{tt+1}}, \mathrm{IL}_{\mathrm{tt+1}}, \mathrm{O}_{\mathrm{tt+1}}, \mathrm{D}_{\mathrm{tt+1}}]_{\mathrm{tt=t-k+1}}^{\mathrm{t}} \qquad (4\text{-}12)$$

Also, since the above equation considers only the last m observed states, the considered environment is a partially observable Markov decision process. In addition, ϵ-greedy is implemented in order to trade-off between exploration and exploitation. ϵ is the percentage of time steps in which agent takes an action randomly rather than taking the action based on the minimum reward (Figure 4.5). Although each state is a frame of input parameters with size k (Eq. (4-12)), to make Figure 4.5 simpler and more understandable, a general form of the figure without framing is displayed instead of showing a frame of inputs.

4.4.2 DNN Section of Algorithm

The *DNN* is applied so as to find a function approximation of *RL*. *DNN* can be utilized instead of linear, kernel methods, or general neural networks. Direct training based on complex inputs is feasible in *DNN*. Features are extracted from one state and a function approximator uses these features as well as certain parameters to extract the cost to go to next state (see Figure 4.3). An approximator is essential due to the huge size of states and this approximation is conducted with a *DNN*. The inputs and outputs of this *DNN* are states and Q-value of actions, respectively. As a result, the size of output is equal to the number of possible actions. This leads to some limitations in the cardinality of action space, though there is no limitation on action space in the theory. The general structure of *DL* section containing k frames of states is shown in Figure 4.4, in which a state is a set of interrelated features. The figure displays that the output of *DNN* is x, which is the difference between order and demand. One parameter is the next demand whose value is observable at the beginning of the next time. Then, the next ordering value is calculated. Also, *IT*, *OO*, and *IL* are updated based on the relations given in the previous sections (Eqs. (4-3) to (4-7)). These three parameters as well as their corresponding ordering value and demand are used as the inputs of the next state.

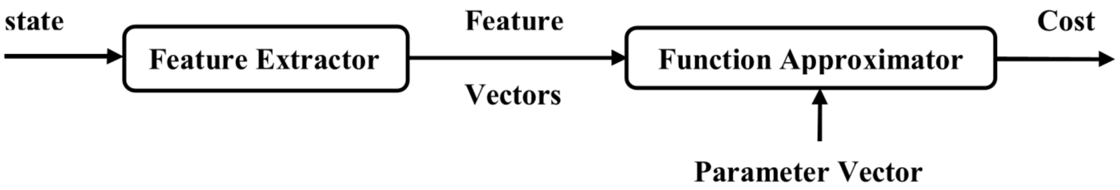

Figure 4.3 Function approximation based on feature extraction

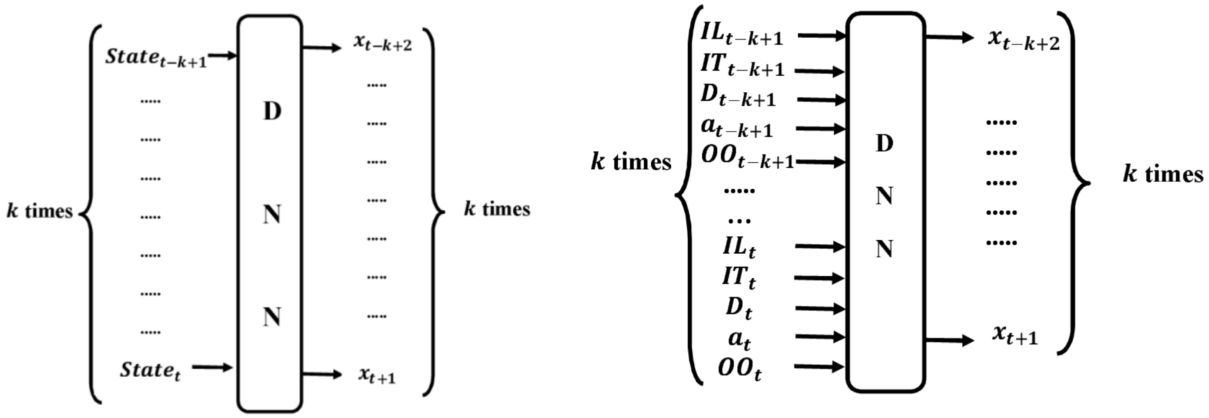

Figure 4.4 The general input/output of *DL* approach for one agent used to find estimation of difference between order and demand ($x = O - D$) based on features of k current states

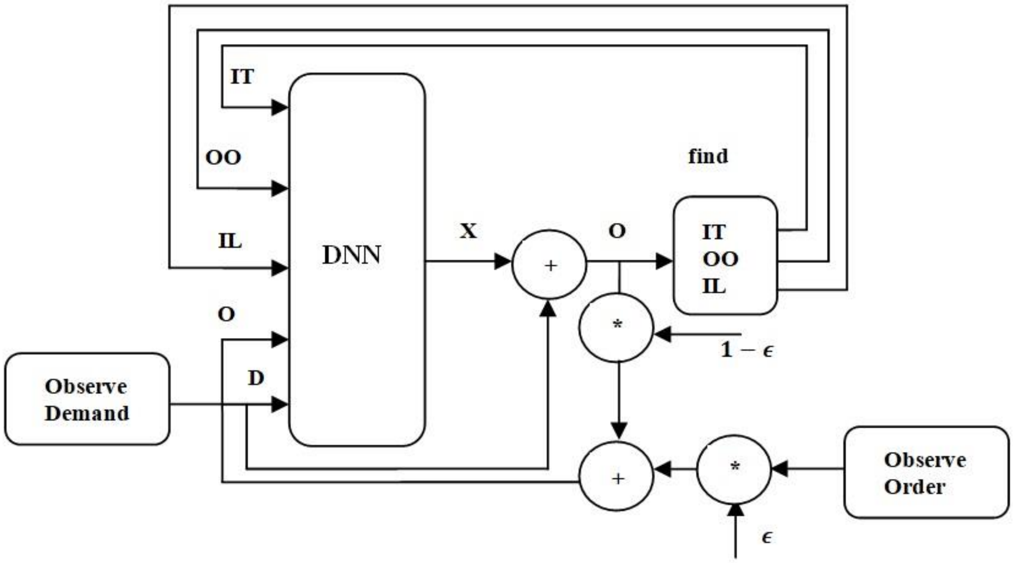

Figure 4.5 The general structure of ϵ-greedy with *DNN* to update the state of one agent

The parameters of the next state S_{t+1} are $IT_{t+1}, OO_{t+1}, IL_{t+1}, O_{t+1}$ and $\mathrm{D}_{\mathrm{t+1}}$. Finally, if the holding, shortage, and ordering coefficient costs are available, the cost of time step can be calculated by considering the inventory level and ordering values (Eqs. (4-8) to (4-9)).

4.4.3 Implementation of Experience Replay

Experience Replay (*ER*) is an important method contributing to most of the latest advances in *DRL*. In *RL* method, the agent observes a series of experiences and then utilizes these experiences to update their internal beliefs. A tuple of action, reward, current state, and next state could be the current experience and the agent is able to use this experience in order to update the value-function by utilizing TD-learning. After using the current experience for the updates, standard classical RL algorithms ignore the current experience, whereas the recent advanced algorithms take it into account. An experience in standard *RL* algorithms is thrown away after being utilized for an update. Recent advances in *RL* introduce *ER*, a method which stores experiences in a memory buffer with a constant size. Since the size of buffer is constant, when this buffer is full and new experiences are observed, the oldest experiences in memory buffer are discarded. In every time step, sampling of a random batch of experiences from buffer is made so as to update the parameters of the agent.

In order to remove the correlation in sequence (transition between two states), reduce the overall variance of the transition, and make the variance smoother against a variation in data distribution, *ER* is applied as an improvement for the algorithm. Consequently, $< s_t, a_t, r_t, s_{t+1} >$ is considered as the current experience e_t which is used for selecting the optimized behaviour. Instead of running the simulation based on state and action, the system saves a set of state, action, reward, and next state. To enjoy the benefits of batch normalization, a batch with size b of *ER* is implemented. Also, learning section and gaining experience need to be combined so as to improve the policy. This policy causes another behaviour which should explore the near-optimal actions applied for learning. In addition, since the applied loss function is designed for *DQN* without *ER*, it should be revised by feeding $< s_t, a_t, r_t, s_{t+1} >$ into *DQN*.

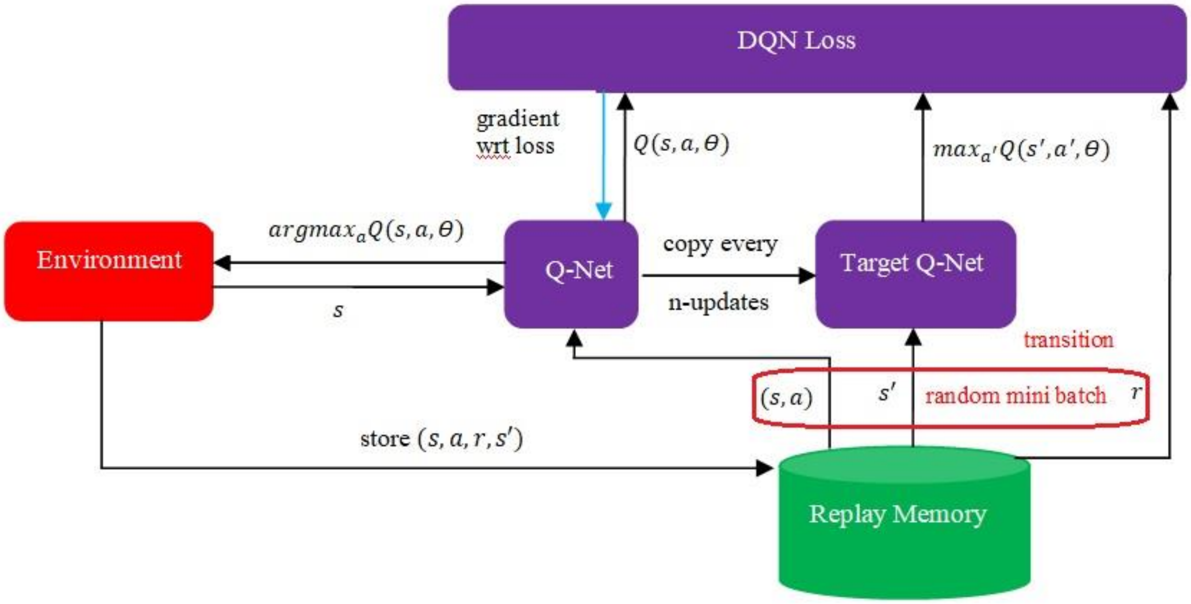

Figure 4.6 The general implementation of *ER* with one agent in each time step

Another point is to maintain two *DNN*s with parameters θ and $\bar{\theta}$ and switch one of them with the other. This assists the stability of the algorithm when a non-linear approximate function is applied. $\bar{\theta}$ defines the alternate frozen version of weighting coefficients. The parameters of target Q-Network are updated every n steps. Figure 4.6 illustrates the structure of *DQN* when *ER* is added to stabilize it. Figure 4.6 shows how to implement *ER* when the cost function is maximized. In this figure, s' is the next state and a' is the next action.

4.4.4 Proposed DQN Algorithm

Algorithm 1 is a *DQN* designed for the above-mentioned inventory control problem and can be utilized in order to find the orders which reduce the overall cost of one inventory agent.

4.5 Hyperparameters Tuning

The *DQN* approach is examined in order to reduce the cost of one-agent (one-stage) inventory control. A number of hyper parameters are checked so as to find the best solution. A list of settings for the main hyperparameters in deep Q-learning is presented in Table 4.3.

Algorithm 1: *DQN* Algorithm for Inventory Control

```
Inputs:
        replay memory size $\mathbf{M}$,
        mini-batch size $\mathbf{B}$,
        greedy parameter $\boldsymbol{\epsilon}$,
        pre-train time steps $\mathbf{pr}$,
        target network update rate $\frac{\mathbf{1}}{\mathbf{n}}$ (update one time of every $\mathbf{n}$ steps),
        discount factor $\boldsymbol{\gamma}$.
Parameters:
        parameters of the primary/target neural network $\boldsymbol{\theta}/\bar{\boldsymbol{\theta}}$,
        replay memory e,
        step number t.
   For episodes = 1 : N {/*N: max episodes*/
        /*initialize*/
        Initialize Experience Replay Memory, $\mathbf{e} = [\ ]$ /*e is a memory with size M of state*/
        [IL, O, OO, d, IT]=[$\mathbf{IL_0}$,0,0,$\mathbf{d_0}$,0] /*starting scenario of state*/
        For t = 1 : T {/*T: max running time step*/
                Observe demand and current state
                /* ϵ greedy algorithm: exploration vs exploitation */
                $\mathbf{a_t} = \epsilon \times \mathbf{random}(\mathbf{a_t}) + (1-\epsilon) \times \mathbf{argmin_a Q}(\mathbf{s_t}, \mathbf{a}; \boldsymbol{\theta})$
                Execute action $\mathbf{a_t}$, observe $\mathbf{r_t}$, and $\mathbf{s_{t+1}}$
                If memory size > M:
                        Remove oldest from memory
                        Add <$\mathbf{s_t}$,$\mathbf{a_t}$, $\mathbf{r_t}$, $\mathbf{s_{t+1}}$> to e
                If |e| > B and episodes > pr:
                        Select a mini-batch(B) of experiences <$\mathbf{s_t}$,$\mathbf{a_t}$, $\mathbf{r_t}$, $\mathbf{s_{t+1}}$ > from  e
                    /*calculated the Q-function and updating the reward*/
                If episode=T: /*if the final state*/
                        Set $\mathbf{y_j} \leftarrow \mathbf{r_j}$
                Otherwise /*if it is not final state*/
                $y_j \leftarrow r_j + \gamma . \min_a Q(s_j, a_j, \overline{\theta})$
                /*Gradient Descent: FF BP to optimize loss function*/
                /*Run Feed Forward (F.F) and Backward Propagation (B.P)
                Find Mean Square Error (MSE):
                Find Gradient Descent on $(y_j - Q(s_j, a_j, \theta))^2$ */
                /*updating the weight of NN*/
                update one time of every n iterations
               update ϵ
         }/* for t = 1 : T */
    } /* for episodes = 1 : N*/
```

Table 4.3 Main hyperparameter values

Hyperparameters	**Value**
Mini-batch size	32
Replay-memory size	100000
Agent history length (frame)	5
Learning rate	0.001
Discount factor	0.995
Exploration decay	0.95
Initial exploration	1
Final exploration	0.01
Loss Function	MSE

4.5.1 Reward, Inputs/Outputs and Hidden Layers of DQN

Inventory level *IL*, inventory transit *IT*, ordering value *O*, on-ordering value *OO*, and demand *D* are five main parameters of inventory control constituting a state which is an input into *DNN*. In fact, a frame of these five main parameters of inventory control and the difference between ordering value and demand are the inputs and output of a *DNN*, respectively. In addition to the input and output layers, the *DNN* network consists of three hidden layers including 135, , 80, and

50 nodes. The network is Fully Connected (FC) with an activation function except for the last layer which obtains a linear activation function. DNN is used as a function approximator whose output layer's nodes are related to a possible action. As a result, the number of nodes in the output layer is selected by the action space which follows the Kimbrough's rule and the difference between ordering value and its corresponding demand is found. Also, reward (cost) of each time step is calculated based on inventory level, backorder, and a constant value if there is an order. The ordering value is the summation of the current observed demand and approximate action. This approximation of action is made with a DNN whose input is state. The related relations are presented in the primary sections of this chapter.

4.5.2 Frame and Batch Size

Since the results of consecutive steps are correlated, the size of frame is determined by considering the summation of lead-time of making an order and lead-time of receiving it. In both cases, the lead-time is considered to be equal to two. As a result, any frame size greater than four seems to be appropriate. Therefore, in this research project, the frame size is set to five (Table 4.3). Nevertheless, sometimes the latency of observing the effect of one change in inventory level may be greater than the above-mentioned summation. This problem is resolved by ER, which was demonstrated in the previous sections. In order to eliminate the correlation between observation and reduction of the output variance, ER is applied and a mini-batch is chosen in each training step. The mini-batch selects a batch of actions from the starting point until now because the effect of selecting an action may be seen with a delay of several steps. On the other hand, it is also important that the batch size be large enough so as to eliminate the observation of noisy loss function. This is because small batch size makes the loss function noisier. However, it cannot be set to a very large number due to time complexity. The batch size selected in this research is 32.

4.5.3 Activation Function and Type of Different Layers

DNN is utilized to select action and the most common activation function for DNN is $ReLU$. $ReLU$ is demonstrated by $h = max(0, a)$, where $a = Wx + b$. The major benefits of $ReLU$ are training network fast, being sparse and reducing the likelihood of vanishing gradient. Since $ReLU$ is mostly linear and zero for all positive and negative values, it does not have a complex formula, and as a result, it does not take a long time to train or run in comparison with Sigmoid or Tanh. The $ReLU$ speed of convergence is high and changes linearly mostly. Also, the likelihood reduction of the vanishing gradient arises when $a > 0$ and the gradient is constant while the gradient of $Sigmoid$ goes down as the absolute value of x goes up. The constant gradient of $ReLU$ provides training fast. In addition, the $ReLU$ is more sparse when $a \leq 0$. In contrast, $Sigmoid$ is mostly a non-zero value, which results in dense representations.

The *DN*N is a *FC* neural network and the activation function is chosen as given below. In order to solve the dying problem of $ReLU, Leaky\ ReLU$ is preferred. For instance, a large gradient flowing through $ReLU$ could make updating the weight difficult. $Leaky - ReLU$ tries to solve the dying problem of $ReLU$. To achieve this, $Leaky - ReLU$ has a small negative slope (~ 0.1) instead of being zero for negative input. $Leaky - ReLU$ function is written as $f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)$, where α is a small constant. Another benefit of $Learky - ReLU$ is to be more balanced by keeping the mean activation close to zero and probably the speed of learning is greater than $ReLU$. The activation function suffers from inconsistency. This problem is somehow solved with parametric $ReLU$, which is $Leaky - ReLU$, whose α is variable. Therefore, this activation function is utilized for all of the layers except for the last layer whose activation function is linear.

4.5.4 Loss Function and Optimizer

Mean Absolute Error (*MAE*), Mean Square Error (*MSE*), Huber and *Log-Cosh* are different regression loss functions verified in order to select the best loss function. Since *MSE* pays more attention to large errors in comparison with *MAE*, it is chosen as the loss function. In addition, similar to DeepMind's paper, the linear approximation is applied to observe the effect of using *DQN*. For comparison, the *DNN* is replaced by a linear approximation of input layer into output layer without considering any hidden layer. *Adam* optimizer adaptively updates the learning rate and also considers both first-order and second-order moments by using the *SGD* procedure. Recently, it is claimed that the proper tuned *SGD* surpasses the adaptive method similar to *Adam* (Keskar and Socher 2017). However, *Adam* is still selected because it is practically popular in Q-learning with a function approximation (Lillicrap et al. 2016; Mnih et al. 2016). A larger number of hyperparameters for *SGD* makes its proper tuning harder, and therefore, *SGD* need more training due to a lower speed. Also, since the speed of convergence of *Adam* optimizer is fast and it is an adaptive approach attaining acceptable overall performance in comparison with other back propagation optimization approaches (Kingma and Ba 2015), *Adam* optimizer is selected.

4.5.5 Size of ER Memory, Updating Frequency, Learning Rate and ϵ

The size of memory of ER is 100,000 and the system is running over 100,000 episodes. The parameters of the DNN are updated and saved every n iteration. Different amounts including 500, 5000, and 10000 for updating the parameters of DNN are evaluated and finally n is chosen as 5000. Training starts at step 300 to observe the behaviour of system before training. The learning rate is low due to getting rid of dying $ReLU$. For instance, if the learning rate is too high, it can be seen that a large percentage of neurons never be activated during the whole training dataset. When the learning rate is set properly, the problem of dying $ReLU$ is less frequently observed. The learning rate is set to 0.001 and ϵ decreases from 1 to 0.01 with a decay rate of 0.995.

4.5.6 Running Environment and Setting Parameters

The program is coded in Python 3.6 with *Tensorflow* 1.10.0 and *Keras* 2.1.6. The code is executed on Compute Canada allocated one GPU and 64 GB memory. The running times are different based on the computations needed by different algorithms and their settings. The lead-time is set to a constant value equal to two and demand is considered to be randomly selected among [1,2,3].

4.6 Experiments and Discussions

The selection of the benchmark algorithm could be considered from two different points of view. On the one hand, most of the last works on inventory control optimization with *RL* were compared with a type of *S* policy (Van Roy et al. 1998, Giannoccaro et al. 2002). On the other hand, *DQN* approach presented by DeepMind (Mnih et al. 2015) was compared with a linear function approximator, a disable/enable experience replay or a target Q-network, and a professional human games player (Mnih et al. 2015). The stock-out rate is an essential parameter to calculate classic <*s*,*S*> and <*R*,*Q*> policy. It is not considered directly in the present cost function, whereas it affects indirectly the cost function based on the ratios of the different cost coefficients. Since the demand is randomly selected among [1,2,3], it does not follow the distribution such as normal or is combined with some noise, whereas the classical *<s,S>* and *<R,Q>* policies are designed based on predefined demand distribution. Therefore, their classical versions are inappropriate to be a baseline. Van Roy et al. (1998) performed the exhaustive search to determine the best order-up to level. In order to make the *<s,S>* and *<R,Q>* policies more trustable baselines, the inputs data of *DQN* which are the demands are saved. Then, the optimal values of pairs of *<s,S>* and *<R,Q>* are extracted by a grid search on the input data. However, there is no pre-knowledge of the input data for *DQN*. In addition, since in most of selected case studies, the results of the *<R,Q>* policy are slightly better than those of the *<s,S>* policy, the *<R,Q>* policy is used as a baseline to evaluate the performance of *DQN*. Since the ratio of the coefficient of backorder cost to that of the holding cost was considered to be two in several previous works following Sterman (1989), this ratio is utilized in case study 2. The results of the case study whose $<C_h, C_p, C_0>$ coefficients are *<1,100,20>*, are illustrated in the following.

To evaluate the performance of *DQN*, the average of long-run system cost for some algorithms and different settings of several case studies are compared. Also, the fluctuation of the different parameters is studied to observe the behaviour of each case study. The average reward of *DQN* reasonably decreases during training while suffering from the instability demonstrated by the fluctuations of average reward. This instability might be attributed to catastrophic forgotten (McCloskey and Cohen 1989) which happens by overwriting new training samples, which leads to losing the stored information. Several settings for the parameters of different algorithms were examined to address this problem. To alleviate this instability, experience replay is implemented, which somehow mitigates the stability issues.

Also, the performance of two different regression metrics Mean Square Error (*MSE*) and Mean Absolute Error (*MAE*) are compared in Figure 4.7 (Left). The result of each point is produced by calculating the average of overall costs in every 1000 episodes. As displayed in this figure, both of them start with a large deviation and then gradually level off. However, *MSE* drops sooner than *MAE* and is smoother in the final steps. The cost becomes stable after about 65000 episodes. Since the large cost is undesirable, *MSE* is more useful.

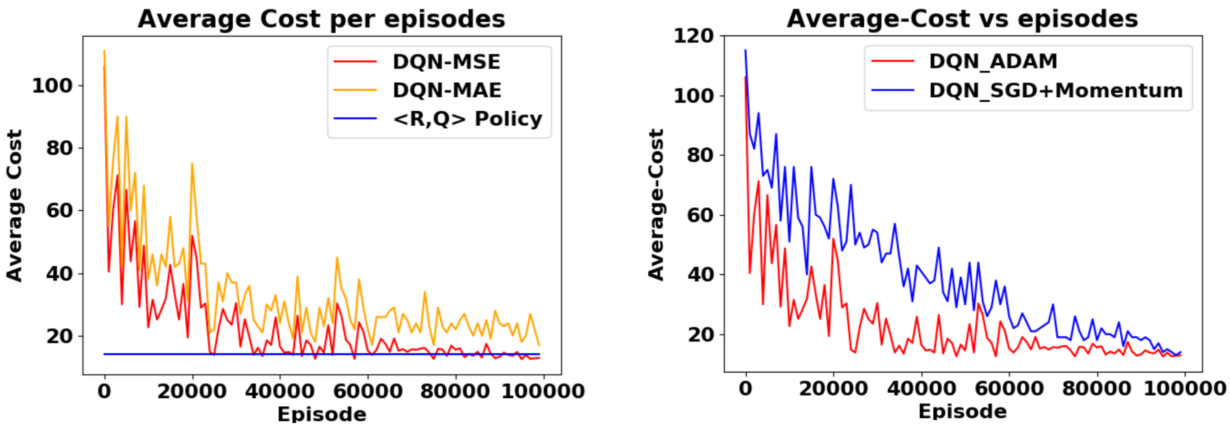

Figure 4.7 Comparison of different regression metrics and different optimizers.

The performance of two main optimizers *ADAM* and *SGD* are compared in Figure 4.7 (Right). The number of tuning parameters for *ADAM* is lower than that for *SGD* with Momentum, which leads to easier tuning the *ADAM* optimizer. It is demonstrated that the performance of *SGD* is worse than that of *ADAM* during the first episodes. However, after passing approximately 55000 episodes, the differences between the results of *SGD* and *ADAM* gradually decrease. This trend continues until the results of *SGD* coincide with those of *ADAM* in the last 7000 episodes.

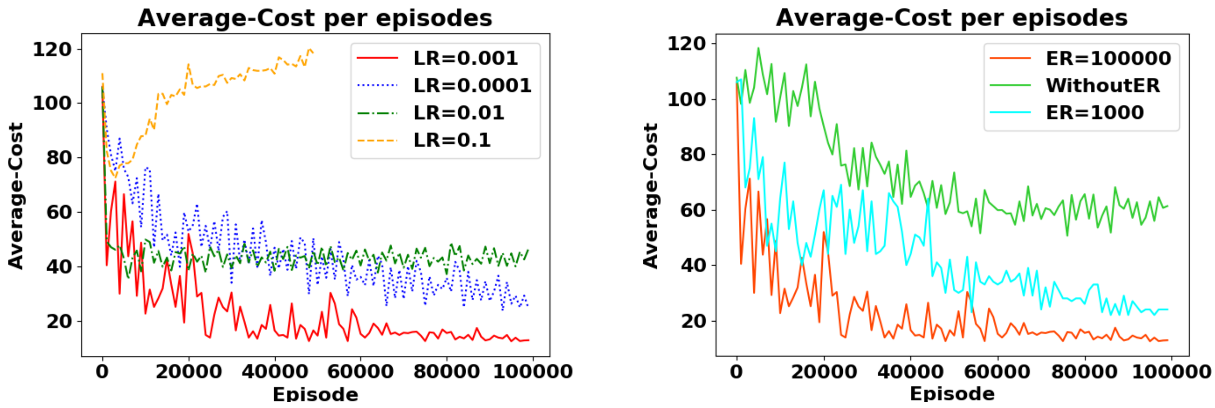

Figure 4.8 Comparison of different values of learning rate (Left) and experience replay (Right)

One of the major considerations during tuning is the learning rate. The effect of different ranges of learning rate are shown in Figure 4.8 (Left). A high learning rate leads to dying the activation function and fast decaying the cost. Consequently, it is unable to settle in an appropriate point. In contrast, a low learning rate results in a small decay and needs much time to reduce cost sufficiently. However, a proper learning rate has a smaller chance of dying in comparison with the results of the low learning rate and has a greater chance to diminish the cost and obtain a near-optimal cost. Nevertheless, the cost is still a little noisy, which might be due to a small batch size. If the learning rate *LR* is too high (i.e. *LR=0.1*), the final cost is very large. The cost decreases with decreasing the learning rate until the rate reaches 0.001. Then, when the learning rate is too low (i.e. *LR=0.0001*), the final cost increases. Therefore, finding the fitted learning rate is an important factor in the performance of the algorithm.

The influence of enabling and disabling *ER* is displayed in Figure 4.8 (Right). The results become smoother with increasing the number of episodes and the final value of cost is lower when the amount of memory of *ER* increases. If *ER* is disabled, the results are not as stable as the case where *ER* is enabled. Also, the effects of enabling/disabling *ER* and target Q-network on the results for different case studies are presented in Table 4.4. It is clearly demonstrated that disabling the target Q-network and specially the replay memory has detrimental impacts on the algorithm performance. This is because by random selection of the parameters, the correlations in the observation sequence vanish. *ER* benefits from the ability to improve the data efficiency and makes the training more stable. *ER* can find the experiences from the previous time, which can be effective when learning is carried out several times. The *DQN* with *ER* leads to a better convergence when the function approximation is trained. This is attributed to the fact that data is assumed to be independent and identically distributed (*i.i.d.)* in most of proofs for the convergence of supervised learning approaches. This ability makes the algorithm more efficient in comparison with the others.

Table 4.4 The influence of replay and separation of the target Q-network

Coefficient	W* Replay, W** target Q	W Replay, WO target Q	WO Replay, W target Q	WO Replay, WO target Q
1-100-5	6.676	8.732	37.042	56.324
1-10-16	9.948	21.942	65.572	67.57
1-100-20	12.582	29.044	49.200	62.336

*W: with, **WO: without

The impact of skipping frame is shown in Table 4.5. The results show the skipping frame is a very effective factor in performance. This is reasonable because there is a delay in observing the influence of an action on the cost function. This delay which is mainly due to availability of lead-time, makes the parameters of consecutive steps dependent. Therefore, it is important to consider a frame of parameters instead of just considering parameters of the current time step.

Table 4.5 Comparison between with/without of skipping frame

Coefficient	With skipping frame	WithOut skipping frame
1-100-5	6.676	66.784
1-10-16	9.948	99.012
1-100-20	12.582	69.662

The overall cost and different parameters of inventory control for case study *<1,100,20>* are compared with other methods such as *<R,Q>*, *<s,S>* policies and linear regression Q-learning in Figures 4.9-11. As displayed in Figure 4.9 (right), the overall cost of *DQN* is appropriate even from the first steps. One interesting point is that the results for *DQN* are suitable, even though the values of different parameters of local information for *DQN* approach do not follow the behaviors of the other famous techniques. The results demonstrate that the range of step-cost and inventory level for *DQN* are proper compared to the other techniques (Figures 4.10 and 4.11).

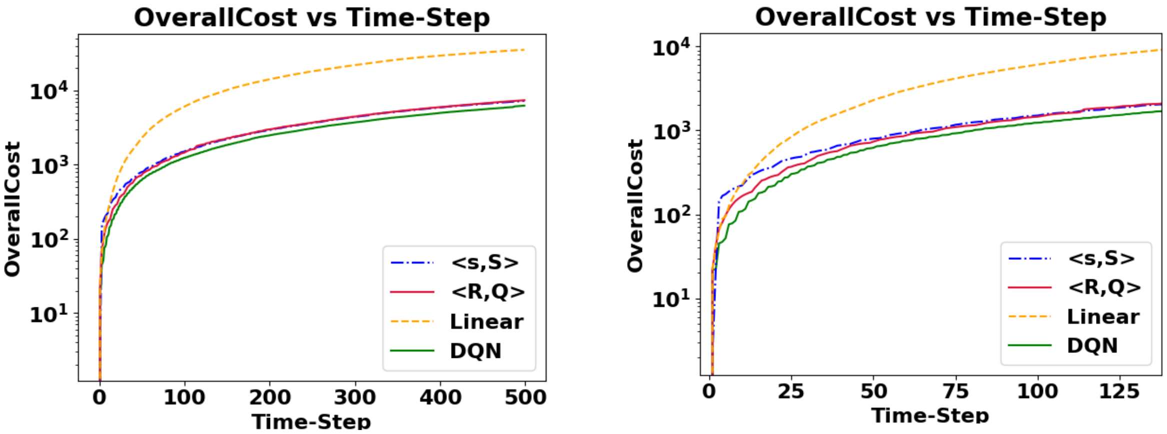

Figure 4.9 Overall cost of different methods (left: all time-steps, right: first 125 time-steps)

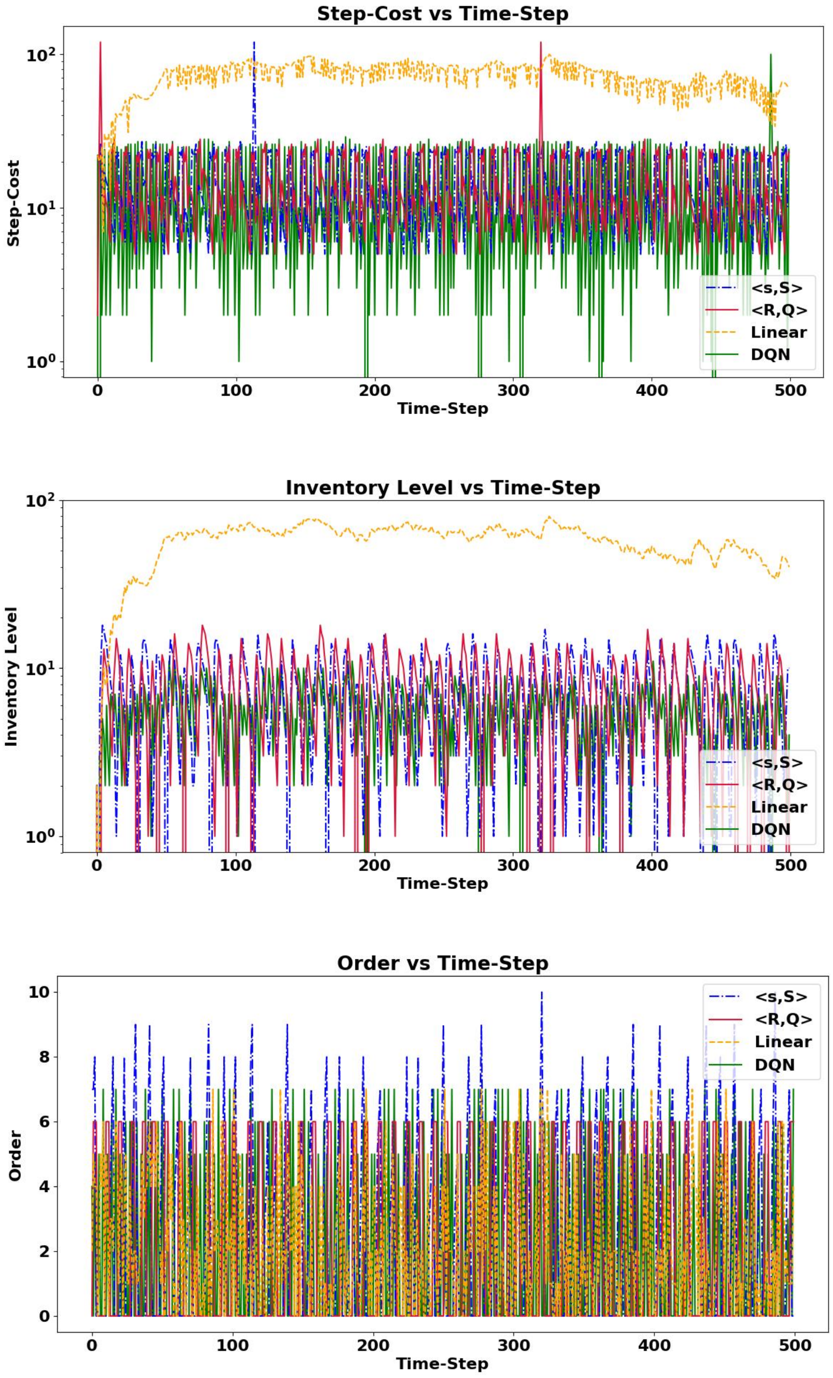

Figure 4.10 Step-cost, *IL*, and *O* of different methods

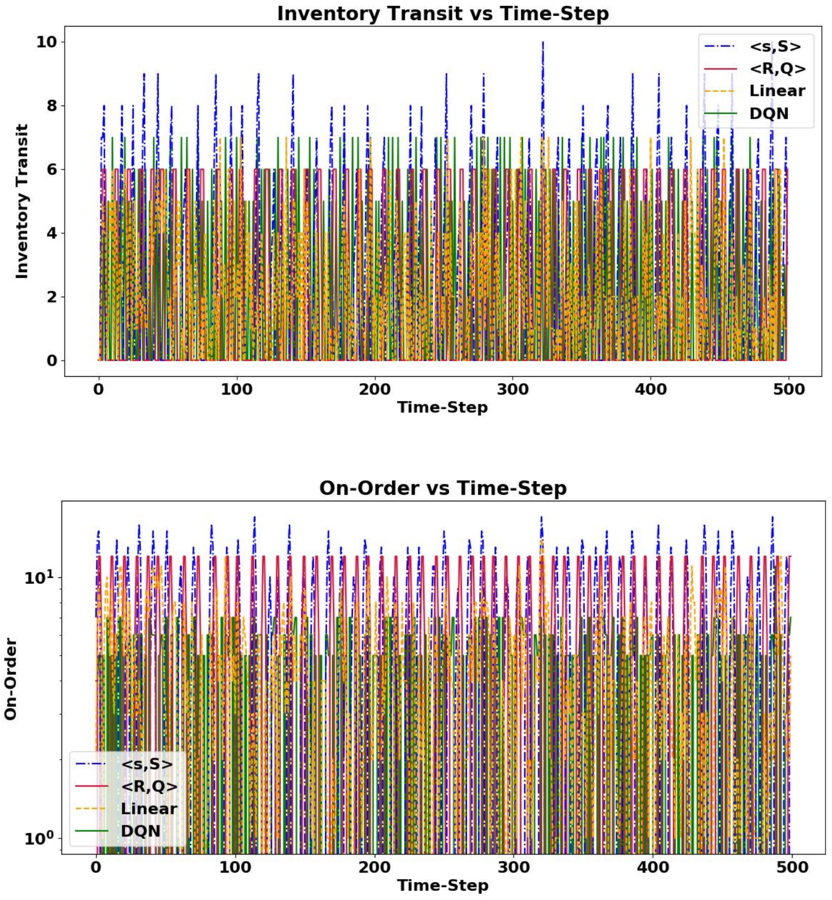

Figure 4.11 *IT* and *OO* of different methods

The fluctuations of main parameters of these approaches are compared (Figures 4.10 and 4.11). The number of times of stock-out for *DQN* is equal to and less than those of <*s*,*S*> and <*R*,*Q*> policies, respectively (see the step-cost in Figure 4.10). Since the second coefficient is one hundred times greater than the first cost coefficient and five times higher than the third cost coefficients, the near-optimal solution is obtained with a few number of stock-out. The average cost of *DQN* are compared for some algorithms such as linear Regression Q-learning (*RQL*) which is an algorithm by DeepMind Company (Mnih et al. 2013) used for evaluating the performance of *DQN* in Atari games. Its structure is similar to *DQN* and the only difference is the omission of deep layers. The other comparisons are made with famous inventory management policies such as constant <*R*,*Q*> and <*s*,*S*> policies. The performance of the algorithm is acceptable compared with other approaches.

The *DQN* algorithm is evaluated for different cost coefficients in Table 4.6. The different cost ratios are chosen in terms of the real values as well as the values whose level of stock-out for their optimized solution are very low. Overall, the present *DQN* selects appropriate actions in comparison with the <*s,S*>, <*R,Q*> policies and regression *RL*. The number of times of stock-out for *DQN* for case studies 1 to 8 is 348, 199, 0, 35, 39, 1, 0, and 34, showing that there is a few number of times of stock-out when the second coefficient is higher than the others. As shown in Table 4.6, the performance of *DQN* is satisfactory, while Regression Q-learning is the worst case except for the coefficients of *<1-50-2>* whose result is in the second place. In addition, the gap between the *DQN* results and the best results of the other methods is calculated and given in Table 4.6.

Table 4.6 Comparison of average cost for different coefficients and techniques

No.	**Coefficient**	**RQL***	**DQN**	**<R,Q>**	**<s,S>**	**Gap**
1	1-1-1	17.148	2.034	2.382	3.836	-14.6%
2	1-2-5	26.162	4.644	5.334	6.194	-12.9%
3	1-100-5	48.696	6.676	8.328	8.248	-19.0%
4	1-17-27	95.248	11.948	15.982	16.042	-25.2%
5	1-10-16	113.204	9.47	11.736	12.07	-19.3%
6	1-100-20	71.33	12.582	14.574	14.87	-13.6%
7	1-50-2	4.886	3.96	6.234	6.898	-18.9%
8	1-5-8	29.894	6.898	7.604	8.466	-9.2%

*RQL: Regression Q-learning

CHAPTER 5 SUMMARY, CONCLUSION, FUTURE WORKS, AND RECOMMANDATIONS

The recent technological advancements provides huge data generation. It is often challenging to deal with these large volumes of data. To handle huge data generation, some research areas such as speech recognition utilize *ML* algorithms and especially *DL* techniques. In contrast, most of product manufacturing problems such as inventory control are currently solved by imposing constraints to the datasets. In order to solve the inventory control problem handling enormous raw datasets, a data-driven *ML* technique is implemented in the present research.

The present inventory control problem aims to reduce the long-run overall cost which is obtained by finding orders based on input demands. The overall cost is the summation of linear holding (inventory on-hand), linear shortage (unmet demand), and fixed ordering (each time of ordering) costs. The above-mentioned inventory control problem for single-agent is solved to provide an insight into sequential multi-agent inventory control problems, which are hard to be put into practice as most of their solutions highly need many details about local and communicated information while the data is not available. For instance, the inventory capacities should be pre-known and limited or discretized if they are unrestricted, although even the best discretization may lead to losing the precision. Also, in reality, the agents do not share their individual information in *POMDP*s or even a single agent is a *POMDP* as it considers a part of observation of its local information due to some limitation when implemented. This property makes the past *RL* methods unusable. In contrast, the present algorithm not only does not need to know the constraints on the individual information such as inventory capacity, but it is also able to solve a *POMDP* environment which does not have access to the whole individual data.

The proposed approach in the present research is a type of *DRL* called *DQN* which can solve the problem by employing both *RL* and *DL* even the environment is a Partial Observable (*PO*). Also, since there are some latencies, lead-time related parameters such as on-order inventory and inventory transit in addition to inventory level, demand, and orders are considered as the parameters of state, which is useful. *MDP* is able to model uncertain decision making problems, while the *DL* part of *DRL* brings an ability to *MDP*s so as to resolve problems of a larger size of states. *RL* is applied for reduction of the overall cost based on making decision about action, i.e.

order, while its state is individual information of an inventory. In addition, *DL* is used to learn ordering minus demand based on the state which is a combination of local information. *DQN* is an off-policy and on-line learning method, and as a result, it obtains not only to learn the environment with any type of input demand distribution, but also to learn the *PO* environment with an unlimited or unknown range of the individual information such as inventory level and vast state spaces (e.g. long-run environments or unlimited capacity).

To be more precise, since successful *RL* policies directly learn from inputs, *DQN* provides the ability of being independent of some details of local information such as inventory level. Therefore, there is no limitation on the state spaces, which is one of the shortcomings of the available *RL*. Also, *DQN* is efficient for the *PO* environment whose agents do not see some parts of their local information or that of the other agents, while this condition occurs in some cases such as in multi-agent environments. In addition, online learning capability of *DQN* makes it able to learn even the type of input demand distributions is unknown, whereas the well-known method such as normal <*s,S*> policy is only desired in the normal distribution demand.

A frame of states is considered due to the probable effects of recent states on the current state. An amelioration is a batch of combination of action, two consecutive states, and reward, where the batch is made of a random selection of experiences from the starting point rather than the batch made of consecutive states. A memory buffer with a certain size containing two consecutive states and related action and reward is called experience replay, leading to a more stable algorithm. Also, different values of some hyperparameters or disabling/enabling these hyperparameters are studied to examine their impacts on the overall cost and stability for some case studies. Since the stability is very critical, *DQN* provides a stable solution to deep value-based *RL*. The stability is investigated by *ER* to break the correlations in data, bring them back to *i.i.d* input data, and to learn from all past policies. In addition, freezing the target Q-network is applied in order to avoid oscillations and break correlations between Q-network and target. Based on the results obtained for several case studies, it is found that the present method outperforms the linear regression *RL*. Also, the performance of *DQN* is comparable with traditional techniques such as <*s,S*> and <*R,Q*> policies. The present approach can also be extended in future to solve serial sequential decision making (multi-agent) supply chain networks even though they are *PO*.

BIBLIOGRAPHY

Atkeson, C.G., Santamaria, J.C. A comparison of direct and model-based reinforcement learning. Proceedings of the International Conference on Robotics and Automation (ICRA) (1997) 3557-3564.

Bertsekas, D.P., Tsitsiklis, J. Neuro-Dynamic Programming. Athena Scientific, Belmont, MA, (1996).

Bruin, T., Kober, J., Tuyls, K., Babuska, R. The importance of experience replay database composition in deep reinforcement learning. Deep Reinforcement Learning Workshop, NIPS, (2015).

Chaharsooghi, S.K., Heydari, J., Zegordi, S.H. A reinforcement learning model for supply chain ordering management: An application to the beer game. Decision Support Systems 45 (2008) 949-959.

Chen, F., Zheng, Y.-S. Lower bounds for multi-echelon stochastic inventory systems. Management Science 40(11) (1994) 1426-1443.

Clark, A.J., Scarf, H. Optimal policies for a multi-echelon inventory problem. Management science 6(4) (1960) 475-490.

Claus, C., Boutilier, C. The dynamics of reinforcement learning in cooperative multi agent systems. AAAI/IAAI (1998) 746-752.

Cox III, J.F., Walker II., E.D. The poker chip game: A multi-product, multi-customer, multi-echelon, stochastic supply chain network useful for teaching the impacts of pull versus push inventory policies on link and chain performance. INFORMS Transactions on Education 6(3) (2008) 3-19.

Croson, R., Donohue, K. Behavioural causes of the bullwhip effect and the observed value of inventory information. Management Science 52 (3) (2006) 323-336.

Devika, K.L., Jafarian, A., Hassanzadeh, A., Khodaverdi, R. Optimizing of bullwhip effect and net stock amplification in three-echelon supply chains using evolutionary multi-objective metaheuristics. Annals of Operations Research 242(2) (2016) 457-487.

Edali, M., Yasarcan, H. A mathematical model of the beer game. Journal of Artificial Society Simulation 17 (2014) 2-14.

François-Lavet, V., Henderson, P., Islam, I., Bellemare, M.G., Pineau, J. An Introduction to Deep Reinforcement Learning. Foundations and Trends in Machine Learning 11(3-4) (2018), DOI: 10.1561/2200000071.

Gallego, G., Zipkin, P. Stock positioning and performance estimation in serial production-transportation systems. Manufacturing & Service Operations Management 1 (1999) 77-88.

Giannoccaro, I., Pontrandolfo, P. Inventory management in supply chains: A reinforcement learning approach. International Journal of Production Economics 78(2) (2002) 153-161.

Goodfellow, I., Bengio, Y., Courville, A. Deep Learning, MIT Press, (2016).

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., Silver, D. Rainbow. Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298* (2017).

Hieber, R., Hartel, I. Impact of SCM order strategies evaluated by simulation-based beer-game. Production Planning and Control 14 (2003) 122-134.

http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Talks_files/deep_rl.pdf

http://www.cs.cmu.edu/afs/cs/academic/class/15780-s16/www/slides/rl.pdf

Ioffe, S., Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:*1502.03167 (2015).

Kaelbling, L.P., Littman, M.L., Moore, A.W. Reinforcement Learning: A Survey. Journal of Artificial Intelligence Research 4 (1996) 237-285.

Katanyukul, T., Chong, E.K.P. Intelligent Inventory Control via Ruminative Reinforcement Learning. Journal of Applied Mathematics (2014) Article ID 238357, 8 pages.

Keskar, N.S., Socher, R. Improving generalization performance by switching from Adam to SGD. *arXiv preprint arXiv:*1712.07628 (2017).

Kimbrough, S.O., Wu, D.-J., Zhong, F. Computers play the beer game: Can artificial agents manage supply chains? Decision support systems 33(3) (2002) 323-333.

Kingma, D.P., Ba, J.L. ADAM. A method for stochastic optimization *arXiv preprint arXiv:* 1412.6980 (2015).

Lee, H., Padmanabhan, V., Whang, S. Information distortion in a supply chain: The bullwhip effect, Management Science 43(4) (1997) 546-558.

Liang, X., Du, X., Wang, G., Han, Z. Deep reinforcement learning for traffic light control in vehicular networks, IEEE Trans. Veh. Technol., *arXiv preprint arXiv:* 1803.11115 (2018).

Lillicrap, T.P., Hunt, J.J, Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D. Continuous control with deep reinforcement learning. International Conference on Learning Representations (ICLR) *arXiv preprint arXiv:*1509.02971 (2016).

Lin, L.J. Reinforcement Learning for Robots Using Neural Networks. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA (1993).

Li, Y. Deep reinforcement learning: An overview. *arXiv preprint arXiv:*1701.07274 (2017).

McCloskey, M., Cohen, N.J. Catastrophic interference in connectionist networks: The sequential learning problem. Psychology of learning and motivation 24 (1989) 109-165.

Melo, F.S., Ribeiro, M.I. Q-learning with linear function approximation. International Conference on Computational Learning Theory (2007) 308-322.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:*1312.5602 (2013).

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wiersta, D., Legg, S., Hassabis, D. Human-level control through deep reinforcement learning. Nature 518(7540) (2015) 529-533.

Mnih, V., Puigdomenech Badia, A., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. International Conference on Machine Learning (ICML) (2016) 1928-1937.

Narottambhai Patel, M., Tandel, P. A Survey on Feature Extraction Techniques for Shape based Object Recognition IJCA 137(6) (2016) 16-20.

Parashkevov, I. Joint Action Learners in Competitive Stochastic Games. Thesis, Cambridge, Massachusetts, (2007).

Puterman, M.L. Markov Decision Processes-Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., New York, NY, 1994.

Schaul, T., Quan, J., Antonoglou, I., Silver, D. Prioritized experience replay. International Conference on Learning Representations. *arXiv preprint arXiv:*1511.05952 (2016).

Shang, K.H., Song, J.S. Newsvendor bounds and heuristic for optimal policies in serial supply chains. Management Science 49 (5) (2003) 618-638.

Silver, E.A. Inventory Management. An Overview, Canadian Publications, Practical Applications and Suggestions for Future Research. INFORM 46(1) (2008) 15-28.

Soria Olivas, E., David Martin Guerrero, J., Martinez Sober, M., Rafael Magdalena Benedito J., Jose Serrano Lopez, A. Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods and Techniques (2 Volumes), 1st Edition, (2009).

Sterman, J.D. Business Dynamics. Systems Thinking and Modeling for a Complex World. Boston, MA: Irwin/McGraw-Hill (2000).

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

Szegedy, C., Toshev, A., Erhan, D. Deep neural networks for object detection. Advances in Neural Information Processing Systems (2013) 2553-2561.

Sui, Z., Gosavi, A. and Lin. L. A Reinforcement Learning Approach for Inventory Replenishment in Vendor-Managed Inventory Systems with Consignment Inventory. Engineering Management Journal 22(4) (2010) 44-53.

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

Sutton, R.S., Barto, A.G. Reinforcement learning: An introduction. Second edition. MIT Press, Cambridge, (2018).

Tsitsiklis, J., Roy, B.V. An analysis of temporal-difference learning with function approximation. IEEE Trans. Automatic Control 42 (1997) 674-690.

Van Roy, B., Bertsekas, D.P., Lee, Y., Tsitsiklis, J.N. A neuro-dynamic programming approach to retailer inventory management, Technical report, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA (1997).

Van der Pol, E., Oliehoek, F.A. Coordinated deep reinforcement learners for traffic light control. NIPS’16 Workshop on Learning, Inference and Control of Multi-Agent Systems (2017).

Watkins, C.J.C.H., Dayan, P. Q-learning. Machine Learning 3 (1992) 279-292.

Zheng, Y.S. On properties of stochastic inventory systems. Management Science 38(1) (1992) 87-103.