
Deep
Reinforcement
Learning
in
Inventory
Management
Kevin Geevers

Master Thesis

Industrial Engineering and Management

University of Twente

Deep Reinforcement Learning in Inventory Management

By Kevin Geevers

Supervisor University of Twente
dr.ir. M.R.K Mes
dr. E. Topan

Supervisor ORTEC
L.V. van Hezewijk, MSc.

December 2020

Management Summary

This research is conducted at ORTEC in Zoetermeer. ORTEC is a consultancy firm that is specialized in, amongst other things, routing and data science. ORTEC advises their customers on optimization opportunities for their inventory systems. These recommendations are often based on heuristics or mathematical models. The main limitation of these methods is that they are very case specific, and therefore, have to be tailor-made for every customer. Lately, ORTEC's interest is gained by reinforcement learning.

Reinforcement learning is a method that aims to maximize a reward and interacts with an environment by the means of actions. When an action is completed, the current state of the environment is updated and a reward is given. Reinforcement learning is able to maximize expected future rewards and, because of its sequential decision making, a promising method for inventory management.

ORTEC is interested in how reinforcement learning can be used in a multi-echelon inventory system and has defined a customer case on which reinforcement learning can be applied: the multi-echelon inventory system of the CardBoard Company. The CardBoard Company currently has too much stock, but is still not able to meet their target fill rate. Therefore, they are looking for a suiting inventory policy that can reduce their inventory costs. This leads to the following research question:

In what way, and to what degree, can a reinforcement learning method be best applied to the multi-echelon inventory system of the CardBoard Company, and how can this model be generalized?

Method

To answer this question, we first apply reinforcement learning on two toy problems. These are a linear and divergent inventory system from literature, which are easier to solve and implement. This way, we are able to test our method and compare it with literature. We first implement the reinforcement learning method of Chaharsooghi, Heydari, and Zegordi (2008). This method uses Q-learning with a Q-table to determine the optimal order quantities. Our implementation of this method manages to achieve the same result of the paper. However, when taking a closer look at this method, we conclude that it does not succeed in learning the correct values for the state-action pairs, as the problem is too large. We propose three improvements to the algorithm. After implementing these improvements, we see that the method is able to learn the correct Q-values, but does not yield better results than the unimproved Q-learning algorithm. After experimenting with random actions, we conclude that the paper of Chaharsooghi et al. (2008) did not succeed in building a successful reinforcement learning method, but only gained promising results due to their small action space, and, therefore, limited impact of the method. Furthermore, we notice that, with over 60 million cells, the Q-table is already immense and our computer is not able to initialize a larger Q-table. Therefore, we decide to take another method, which is deep reinforcement learning (DRL).

DRL uses a neural network to estimate the value function, instead of a table. Hence, this method is more scalable and often defined as a promising method in literature. We chose to implement the Proximal Policy Optimization Algorithm of Schulman, Wolski, Dhariwal, Radford, and Klimov (2017), by adapting the code from the packages 'Stable Baselines' and 'Spinning up'. We define the same hyperparameters as in Schulman et al. (2017) and define the case-specific action and state space ourselves. Next to that, we have decided to use a neural network with a continuous action space. This means that the neural network does not output the probability of a certain action, but outputs the value of the action itself. In our case, this value will correspond to the order quantity that has to be ordered. We chose this continuous action space, as it is more scalable and can be used on large action spaces.

Results

With our DRL method, we improve the results of Chaharsooghi et al. (2008). We also apply the DRL method to a divergent inventory system, defined by Kunnumkal and Topaloglu (2011). We apply the method without any modifications of the parameters of the algorithm, but notice that the definition of state and action vector are important for the performance of the algorithm. In order to compare our results, we implement the heuristic of Rong, Atan, and Snyder (2017) that determines the near optimal base-stock parameters for divergent supply chains. We run several experiments and see that our deep reinforcement learning method is able to perform better than the benchmark with a small difference.

After two successful implementations of our DRL method, we apply our method to the case of the CardBoard Company. We make some assumptions and simplifications, like the demand distribution and lead times, to be able to implement the case in our simulation. We define several experiments in order to find suitable values for the upper bound of the state and action vector. As benchmark, we reconstruct the current method of CBC. In this method, CBC wants to achieve a fill rate of 98 % for every location. Therefore, we determine the base-stock parameters in such a way that a fill rate of 98% is yielded for every location. In this case, we notice that the method is varying greatly in its results. Some runs are able to perform better than the benchmark, while five out of ten runs are not able to learn correct order quantities. We denote the results of both the best and worst run in the final results, as they are too far apart to give a representative average. The final results are:

Total costs of the DRL method and the corresponding benchmarks. Lower is better.

Case	DRL	Benchmark
Beer game	**2,726**	3,259
Divergent	**3,724**	4,059
CBC	8,402 - 1,252,400	**10,467**

Conclusion and Recommendations

This thesis shows that deep reinforcement learning can successfully be implemented in different cases. To the best of our knowledge, it is the first research that applies a neural network with a continuous action space to the domain of inventory management. Next to that, we apply DRL to a general inventory system, a case that has not been considered before. We are able to perform better than the benchmark on every case. However, for the general inventory system, this result is only gained for three out of 10 runs.

To conclude, we recommend ORTEC to:

- Not start with using deep reinforcement learning as main solution to customers yet. Rather use the method on the side, to validate how the method performs in comparison with various other cases.
- Focus on the explainability of the method. By default, it can be unclear why the DRL method chooses a certain action. However, we show that there are several different ways to gain insights into the method, but these are often case specific.
- Look for ways to reduce the complexity of the environment and the deep reinforcement learning method.
- Keep a close eye on the developments in the deep reinforcement learning field.

Preface

While writing this preface, I realize that this thesis marks the end of my student life. It has been a while since I have started studying at the University of Twente. Moving to Enschede was a really big step and a bit scary. Looking back, the years flew by. Years in which I have made amazing friendships and had the opportunities to do amazing thing.

I had the pleasure to write my thesis at ORTEC. At ORTEC, I had a warm welcome and learned a lot of interesting things. I would like to thank Lotte for this wonderful thesis subject and supervision of my project. Your support and advise really helped me in my research. Without you, I definitely would not have been able to yield these good results. Next to that, I also want to thank the other colleagues of ORTEC. It was nice working with such passionate and intelligent people. I am very happy that my next step will be to start as consultant at ORTEC!

Furthermore, I would like to thank Martijn for being my first supervisor. Your knowledge really helped to shape this research. Thanks to your critical input and feedback, the quality of this thesis has increased a lot. I would also like to thank Engin for being my second supervisor. Your knowledge of inventory management really helped to implement the benchmarks and your enthusiasm helped to finish the last part of this research.

Lastly, I would like to thank everyone who made studying a real pleasure. Special thanks to my fellow do-group, my roommates, my dispuut and my fellow board members. Thanks to you I have had a great time. Finally, I would like to thank my family and girlfriend for their unconditional support and love. I hope you will enjoy reading this as much as I have enjoyed writing it!

Kevin Geevers

Utrecht, December 2020

Contents

List of Figures

List of Tables

1. Introduction

This thesis is the result of the research performed at ORTEC in Zoetermeer, in order to develop a method to optimize the inventory systems of their clients. In this chapter, we will first introduce ORTEC in Section 1.1. Section 1.2 covers the problem description, followed by a description of the case we consider in Section 1.3. With this information, we determine the scope and goal of this research, which is given in Section 1.4. To conclude, Section 1.5 describes the research questions and research framework.

1.1 Context

ORTEC is a company that is specialized in analytics and optimization. The company was founded in 1981 by five students who wanted to show the world how mathematics can be used for sustainable growth in companies and society. Since then, they have become the world's leading supplier of optimization software and advanced analytics. While ORTEC started with building optimization software - also referred to as ORTEC Products - they have also set up a consultancy business unit. This business unit focuses on, amongst other things, supply chain design, revenue management, data science, and forecasting. At the moment, ORTEC has about 1,000 employees working in 13 countries, of which around 200 employees are working for ORTEC Consulting.

One of the departments within ORTEC Consulting is the Center of Excellence (CoE). This department is the place for gathering and centralizing knowledge. Next to gathering existing knowledge within ORTEC Consulting, the CoE also looks for interesting subjects to expand their knowledge. This is done by several teams within the CoE with their own field of expertise, such as the Supply Chain team. Within this team, a research project about Multi-Echelon Inventory Optimization (MEIO) was recently finished, which will be used in this research. This research is initiated to elaborate on the subject of MEIO in combination with machine learning.

1.2 Problem Description

Inventory is usually kept in order to respond to fluctuations in demand and supply. With more inventory, a higher service level can be achieved but the inventory costs also increase. To find the right balance between the service level and inventory costs, inventory management is needed. Inventory usually accounts for 20 to 60 percent of the total assets of manufacturing firms. Therefore, inventory management policies prove critical in determining the profit of such firms (Arnold, Chapman, & Clive, 2008). The current inventory management projects of ORTEC Products are focused on Vendor Managed Inventory (VMI). VMI is used for the inventory replenishment of retailers, which commit their inventory replenishment decisions to the supplier. VMI is usually proposed to resolve the problem of exaggerated orders from retailers (Chopra & Meindl, 2015; Kwon, Kim, Jun, & Lee, 2008). In the future, ORTEC Consulting also wants to do more projects on inventory management, but is interested in the tactical aspects, such as a high-level inventory planning. They want to solve a variety of inventory problem and do not want to focus solely on VMI. Hence, they are looking for promising methods to solve different kinds of inventory problems. They are already familiar with classical approaches such as inventory policies and heuristics but want to experiment with data-driven methods, like machine learning.

Most companies are already using certain inventory policies to manage their inventory. With these policies, they determine how much to order at a certain point in time, as well as how to maintain appropriate stock levels to avoid shortages. Important factors to keep in mind in these policies are the current stock level, forecasted demand and lead time (Axsäter, 2015). These policies often focus on a single location and only use local information, which results in individually optimized local inventories and do not benefit the supply chain as a whole. The reason for not expanding the scope of the policies

is the lack of sufficient data and the growing complexity of the policies. Due to recent IT developments, it has become easier to exchange information between stocking points, resulting in more useable data. This contributed to an increasing interest in Multi-Echelon Inventory Optimization. Studies show that these multi-echelon inventory systems are superior to single echelon policies, as the coordination among inventory policies can reduce the ripple effect on demand (Giannoccaro & Pontrandolfo, 2002; Hausman & Erkip, 1994). Despite the promising results, a lot of companies are still optimizing individual locations (Jiang & Sheng, 2009). Therefore, ORTEC sees promising opportunities in optimization methods for multi-echelon inventory management.

Next to that, ORTEC is curious about methods outside of the classical Operation Research domain, because of the limitations of the current methods. Mathematical models for inventory management can quickly become too complex and time-consuming, which results in an unmanageable model (Gijsbrechts, Boute, Van Mieghem, & Zhang, 2019). To prevent this, the models usually rely heavily on assumptions and simplifications (Jiang & Sheng, 2009), which makes it harder to relate the models to real-world problems and to be put into practice. Another way of reducing the complexity and solving time is the use of heuristics. Unfortunately, these heuristic policies are typically problem-dependent and still rely on assumptions, which limits their use in different settings (Gijsbrechts et al., 2019). Although

Figure 1.1: *An overview of types and applications of machine learning. Adapted from Krzyk (2018).*

these mathematical models and heuristics can deliver good results for their specific setting, ORTEC is, because of their consultancy perspective, especially interested in a method that can be used for various supply chains, without many modifications. For such a holistical approach, reinforcement learning is a promising method that may cope with this complexity (Topan, Eruguz, Ma, Van Der Heijden, & Dekker, 2020).

In machine learning, we usually make a distinction between three types: unsupervised, supervised and reinforcement learning. Every type of machine learning has its own approach and type of problem that they are intended to solve. Figure 1.1 shows these three types of machine learning, areas of expertise and possible applications.

At the moment, ORTEC uses machine learning in several projects and has quite some experience with it. Most of these projects are about forecasting and therefore use supervised machine learning. During these projects, their experience and interest in machine learning grew and they became interested in other applications of this technique. Especially reinforcement learning aroused great interest within ORTEC, because it can be used to solve a variety of problems and is based on a mathematical framework: the Markov Decision Process (MDP). MDPs are often used in Operations Research and therefore well known to ORTEC. We will further explain MDPs in Section 3.1.2.

Reinforcement learning is about learning what to do to maximize a reward (Sutton & Barto, 2018). A reinforcement learning model can be visualized as in Figure 1.2. An agent interacts with the environment in this model by the means of an action. When this action is done, the current state of the environment is updated and a reward is being awarded. Reinforcement learning focuses on finding a balance between exploration and exploitation (Kaelbling, Littman, & Moore, 1996). When exploring, the reinforcement learning agent selects random actions to discover the reward of it, while exploitation is done by selecting actions based on its current knowledge. Another important aspect of reinforcement learning is its ability to cope with delayed rewards, meaning that that the system will not per definition go for the action with the highest reward at the moment, but will try to achieve the highest reward overall. These features make reinforcement learning a good method for decision making under uncertainties.

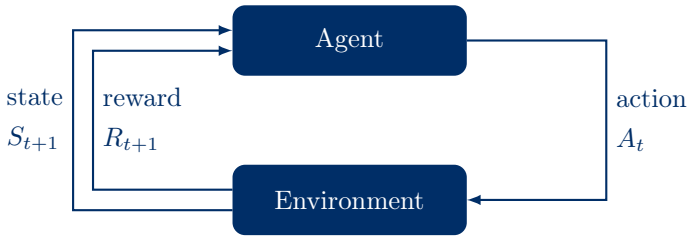


Figure 1.2: *Elements of a Reinforcement Learning system.*

A well-known application of reinforcement learning is in the field of gaming. For example, if we would use reinforcement learning in the game of Pac-Man, our agent would be Pac-Man itself and the environment would be the maze. The action that the agent can take is moving in a certain direction. When Pac-Man has moved into a direction, the environment is updated with this action. A reward is then awarded to Pac-Man; this can either be a positive reward, whenever Pac-Man has eaten a dot, or a negative reward whenever Pac-Man is being eaten by a ghost. However, when Pac-Man first eats the big flashing dot and thereafter eats the ghost, he gets the biggest possible reward. This is an example of a delayed reward. The reinforcement learning system will learn the game by playing it. By exploration, it learns the rewards that are awarded to certain actions in certain states. When the reinforcement learning system plays the game long enough, we will get a reinforcement learning system that excels in playing the game Pac-Man.

Reinforcement learning is developed to solve sequential decision making problems in dynamic environments (Sutton & Barto, 2018). In sequential decision making, a series of decisions are to be made in interaction with a dynamic environment to maximize overall reward (Shin & Lee, 2019). This makes reinforcement learning an interesting method for inventory management. By being able to handle dynamic environments, it can be possible to create a variable inventory policy that is dependent on the current state of the system, instead of a fixed order policy. Sequential decision making and delayed rewards are also relevant for inventory management. In inventory management, actions need to be taken, but the consequences of these decisions are not always directly visible. For example, when a company chooses not to replenish at a certain moment because it still has its items in stock, there is no direct penalty. In a later stage, when the items are out of stock, customers can not be served. Potential sales are lost in this case and it turned out that the company should have chosen to replenish earlier, in order to maximize its profit. Reinforcement learning is able to link a certain reward to every decision in every state and therefore is a promising method to make solid recommendations on when to replenish. Next to that, reinforcement learning could help in solving more complex situations, such as more stochasticity and taking more variables into account. As a result of the earlier mentioned complexity and computation time of mathematical models, multi-item models are scarcely represented in literature at the moment (Chaudhary, Kulshrestha, & Routroy, 2018). With the use of reinforcement learning, it might become easier to use the model for more complex situations. Also, reinforcement learning can include the stochasticity of the demand, whereas, at the moment, only a few models take this into account, as it is really hard to deal with (Chaudhary et al., 2018). Another promising aspect of reinforcement learning is that it could provide a way to solve a diversity of problems, rather than relying on extensive domain knowledge or restrictive assumptions (Gijsbrechts et al., 2019). Therefore, it could work as a general method that requires less effort to adapt to different situations and it will become easier to reflect real-world situations.

The research project about Multi-Echelon Inventory Optimization of Van Santen (2019), introduced a classification concept based on De Kok et al. (2018) to describe the different characteristics of a multi-echelon supply chain. This concept will be further explained in Chapter 2 and will be used to describe the supply chain used in this research.

The features of reinforcement learning sparked the interest of ORTEC and sound very promising for their future projects. This research serves as an exploration in order to find out if reinforcement learning in inventory management can live up to these expectations. In order to build and validate our model, we use the data of a company specialized in producing cardboard, a customer of ORTEC.

1.3 Case Description

The CardBoard Company (CBC) is a multinational manufacturing company that produces paper-based packaging. They are active in the USA and Europe; this case will focus on the latter. CBC has a 2-echelon supply chain that consists of four paper mills, which produce paper, and five corrugated plants, which produce cardboard. Paper mills are connected to multiple corrugated plants and the other way around. An overview of this supply chain is given in Figure 1.3.

The CardBoard Company is interested in lowering their inventory costs, while maintaining a certain service level to their customers. In this research, we use the fill rate as a measure for the service level. The fill rate is the fraction of customer demand that is met through immediate stock availability, without backorders or lost sales (Axsäter, 2015; Vermorel, 2015). At the moment, CBC has a lot of inventory spread of the different stock points, yet they are not always able to reach their fill rate goal. CBC is interested in a better inventory management system that can optimize their multi-echelon supply chain.

To support their supply chain planners, a suiting inventory policy will be determined. The reinforcement learning method will determine the near-optimal order sizes for every stock point, based on their inventory position. In the case of CBC, this results in a policy with which they meet the target fill rate while minimizing the total holding and backorder costs. This inventory policy will provide the supply chain planners a guidance on how much stock to keep on every stock point. In the end, the planners will have to decide for themselves if they will follow this policy or deviate from it.

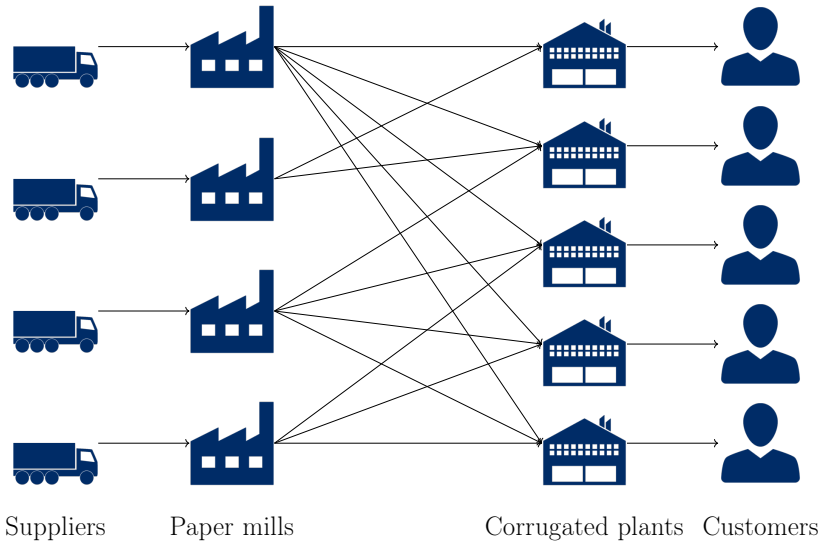


Figure 1.3: *An overview of CBC's supply chain*

1.4 Research Goal

As mentioned before, our research focuses on the usability of reinforcement learning in inventory management. Within ORTEC, inventory management projects are scarce and mostly focused on Vendor Managed Inventory. ORTEC Consulting is interested in high-level inventory planning, wants to solve a variety of inventory problems, and does not want to focus solely on VMI. Therefore, ORTEC wants to see how new methods can be used in inventory management. To make sure this method can be used by ORTEC in future projects, it is applied to the case of the CardBoard Company, a customer of ORTEC. The CardBoard Company wants to find out how their fill rate can be increased and is looking for opportunities to improve their inventory management. The aim of this research is to develop a reinforcement learning method that advises the supply chain planners of the CardBoard Company on the replenishment of different stock points. We build a reinforcement learning method to find out if we can optimize the inventory management. After that, we want to generalize this method, to make sure that ORTEC can also use it at other customers. This leads to the following main research question:

In what way, and to what degree, can a reinforcement learning method be best applied to the multi-echelon inventory system of the CardBoard Company, and how can this model be generalized?

In order to answer this main research question, we formulate several other research questions in the next section. These questions guide us through the research and eventually lead to answering the main question.

1.5 Research Questions

The following research questions are formulated to obtain our research goal. The questions cover different aspects of the research and consist of sub-questions, which help to structure the research. First, we gain more insights into the current situation of the CardBoard Company. We need to know how we can capture all the relevant characteristics of an inventory system and have to define these characteristics for CBC. Next to that, we have to gain insights in the current performance of CBC, such as their current inventory policy and their accomplished service level.

1. **What is the current situation at the CardBoard Company?**
   (a) *How can we describe relevant aspects of multi-echelon inventory systems?*
   (b) *What are the specific characteristics of CBC?*
   (c) *What is the current performance for inventory management at CBC?*

After that, we study relevant literature related to our research. We further elaborate on reinforcement learning and search for current methods of multi-echelon inventory management in order to find a method that we can use as benchmark. Concluding, we look for cases in literature where reinforcement learning is applied in multi-echelon inventory management. For these cases, we list all relevant characteristics that we described in the previous research question. We can then see how these cases differ with CBC. With this information, we choose a qualifying method for our problem.

2. **What type of reinforcement learning is most suitable for the situation of the CardBoard Company?**
   (a) *What types of reinforcement learning are described in literature?*
   (b) *What methods for multi-echelon inventory management are currently being used in literature?*
   (c) *What types of reinforcement learning are currently used in multi-echelon inventory management?*
   (d) *How are the findings of this literature review applicable to the situation of CBC?*

When we have gathered relevant information from the literature, we can begin building our reinforcement learning method. However, reinforcement learning has proven to be a difficult method to implement, and, therefore, it is recommended implement the method on a toy problem first (Raffin et al., 2019). For this, we take two existing cases from the literature and implement this ourselves. When this reinforcement learning method proves to be working, we will expand it in terms of inventory system complexity. This is done by implementing new features in such a way that we work towards the inventory system of CBC.

3. **How can we build a reinforcement learning method to optimize the inventory management at CBC?**
   (a) *How can we build a reinforcement learning method for a clearly defined problem from literature?*
   (b) *How can we expand the model to reflect another clearly defined problem from literature?*
   (c) *How can we expand the model to reflect the situation of CBC?*

While building the reinforcement learning method, we also have to evaluate our method. We will evaluate the method for every problem. Next to that, we will compare the method to see if it does perform better than the current situation of CBC and other multi-echelon methods. We can then gain insights on the performance of the model.

4. **What are the insights that we can obtain from our model?**
   (a) *How should the performance be evaluated for the first toy problem?*
   (b) *How should the performance be evaluated for the second toy problem?*
   (c) *How should the performance be evaluated for CBC?*

(d) *How well does the method perform compared to the current method and other relevant methods?*

Finally, we describe how to implement this model at CBC. Next to that, we discuss how this model can be modified to be able to apply it to other multi-echelon supply chain settings. This way, ORTEC is able to use the reinforcement learning method for their future projects.

5. **How can ORTEC use this reinforcement learning method?**
   (a) *How can the new method be implemented at CBC?*
   (b) *How can the reinforcement learning method be generalized to be used for inventory systems of other customers?*

The outline of this report is given in Figure 1.4. This overview links the research questions to the corresponding chapters and clarifies the steps that will be taken in this research. As mentioned earlier, we will evaluate the performance of the method for every problem. Therefore, the Chapters 4, 5, and 6 cover multiple research questions.

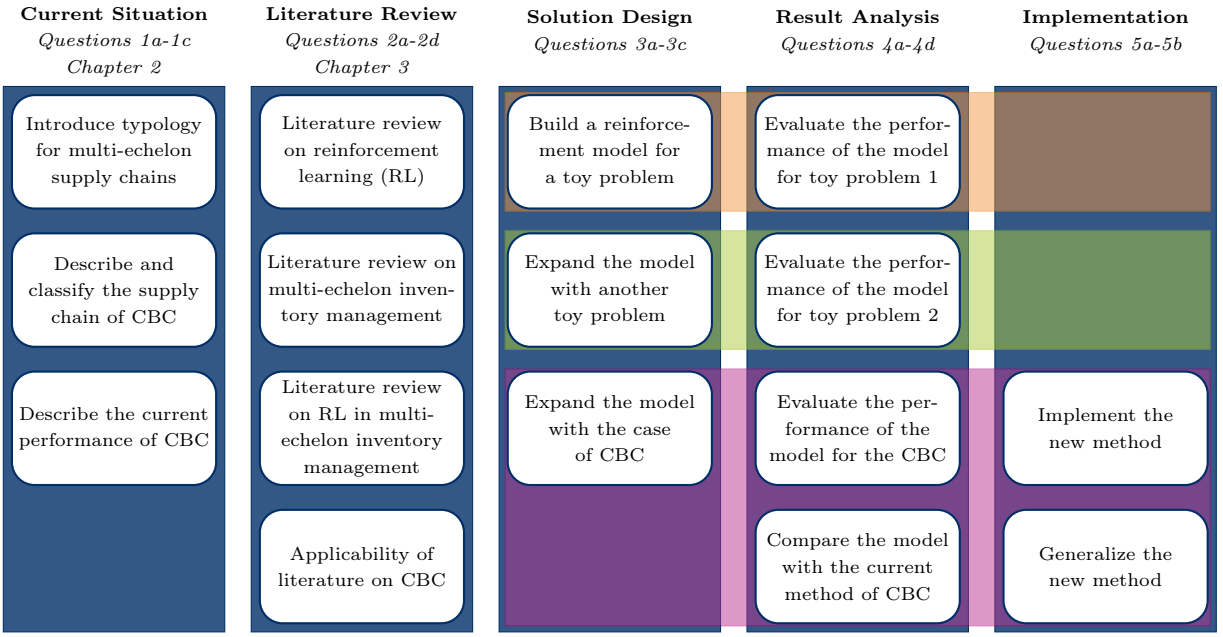


Figure 1.4: *Overview of the research framework*

2. Current Situation

In this chapter, we further elaborate on the situation of the CardBoard Company. We first introduce the classification method that we are going to use in Section 2.1. In Section 2.2, we explain their supply chain in detail and classify it. Section 2.3, contains the current method and performance of the inventory management at CBC. We end this chapter with a conclusion in Section 2.4.

2.1 Classification Method

In order to grasp all the relevant features of the supply chain, we use a typology for multi-echelon inventory systems. This typology was introduced by De Kok et al. (2018) in order to classify and review the available literature on multi-echelon inventory management under uncertain demand. With this typology, De Kok et al. (2018) want to explicitly state all important dimensions of modeling assumptions and to make it easier to link the supply chain problems of the real world to literature. The dimensions stated by De Kok et al. (2018) are based on inventory systems used in literature. Therefore, the classification was extended by Van Santen (2019) to capture all the important aspects of real-world inventory systems. Although it was not possible to ensure that all important aspects of an inventory system are captured in this classification, Van Santen (2019) is confident that the extensions are valuable. Also, their conducted case studies show that the added dimensions were able to capture all the relevant information.

The classification consists of dimensions and features. The dimensions describe the aspects of the supply chain, such as the *number of echelons* and the *distribution of the demand*. The term features is used to define the possible values of these dimensions, for example, *single echelon* is a feature of the dimension *number of echelons*.

Table 2.1 shows all the dimensions and their features. Dimensions that are added by Van Santen (2019) are denoted with a '*'. Next to that, an (S) denotes a dimension or feature that needs a specification in order to simulate the system properly. For example, if a supply chain has bounded capacity, the unit of measure (e.g., kilogram, m^3) needs to be specified (Van Santen, 2019). Further explanation of features that are not described in this thesis can be found in the work of Van Santen (2019).

Table 2.1: *Classification concept (1/3), (Van Santen, 2019)*

Dimension	Values	Feature Explanation
Network specification:		
Echelons	Number of Echelons: a rank within the supply chain network.	
(D_1)	1 (f_1)	Single echelon
	2 (f_2)	Two echelons
	3 (f_3)	Three echelons
	4 (f_4)	Four echelons
	n (f_5)	General number of echelons *(S)*
Structure *(S)*	Relationship between installations.	
(D_2)	S (f_6)	Serial Network, (1 predecessor, 1 successor)
	D (f_7)	Divergent Network, (1 predecessor, n successors)
	C (f_8)	Convergent Network, (n predecessors, 1 successor)
	G (f_9)	General Network, (n predecessors, n successors)
Time	Moments in time where relevant events occur.	
(D_3)	D (f_{10})	Discrete
	C (f_{11})	Continuous

Table 2.1: *Classification concept (2/3), (Van Santen, 2019)*

Part Two Typology, header repeated for readability.

Dimension	Values	Feature Explanation
Information	Level of information needed to perform the computations.	
(D_4)	G (f_{12})	Global
	L (f_{13})	Local
	E (f_{14})	Echelon
*Products	Number of products considered in the inventory system.	
(D_5)	1 (f_{15})	Single Product(-categories)
	n (f_{16})	Multiple Product(-categories) *(S)*
Resource specification:		
Capacity	Restrictions on availability of resources on a single point in time.	
(D_6)	F (f_{17})	Bounded storage and/or processing capacity *(S)*
	I (f_{18})	Infinite capacity
Transportation Delay	Time it takes to deliver an available item.	
(D_7) *(S)*	C (f_{19})	Constant
	E (f_{20})	Exponential
	G (f_{21})	General stochastic
	O (f_{22})	Other
Market specification:		
Demand *(S)*	Exogenous demand distribution for an item.	
(D_8)	C (f_{23})	Deterministic
	B (f_{24})	Compound "batch" Poisson
	D (f_{25})	Discrete stochastic
	G (f_{26})	General stochastic
	M (f_{27})	Markovian
	N (f_{28})	Normal
	P (f_{29})	Poisson
	R (f_{30})	Compound Renewal
	U (f_{31})	Upper-bounded
Customer	Reactions if demand cannot be (completely) fulfilled. (Disservice)	
(D_9)	B (f_{32})	Backordering
	G (f_{33})	Guaranteed Service
	L (f_{34})	Lost Sales
	V (f_{35})	Differs per customer *(S)*
*Intermediate Demand	Defines the echelons that receive exogenous demand.	
(D_{10})	D (f_{36})	Downstream Echelon
	M (f_{37})	Multiple Echelons *(S)*
*Fulfillment	Acceptance of partial fulfillment in case of disservice.	
(D_{11})	P (f_{38})	Partial Fulfillment
	C (f_{39})	Complete Fulfillment
	V (f_{40})	Differs per customer *(S)*
*Substitution	Acceptance of a substitute product in case of disservice.	
(D_{12})	N (f_{41})	None
	I (f_{42})	Accepts substitution more expensive product
	D (f_{43})	Accepts substitution cheaper product
	V (f_{44})	Differs per product *(S)*
Control specification:		
Policy *(S)*	Prescribed type of replenishment policy.	
(D_{13})	N (f_{45})	None
	B (f_{46})	Echelon base stock
	b (f_{47})	Installation base stock
	S (f_{48})	Echelon (s, S)
	s (f_{49})	Installation (s, S)
	Q (f_{50})	Echelon (s, nQ)
	q (f_{51})	Installation (s, nQ)
	O (f_{52})	Other

Table 2.1: *Classification concept (3/3), (Van Santen, 2019)*

Part Three Typology, header repeated for readability.

Dimension	Values	Feature Explanation
*Review Period (D_{14})	Moments the inventory is checked.	
	C (f_{53})	Continuously
	P (f_{54})	Periodically *(S)*
Lot-Sizing (D_{15})	Constraint on replenishment quantity.	
	F (f_{55})	Flexible: no restriction
	Q (f_{56})	Fixed Order Quantity *(S)*
	O (f_{57})	Other *(S)*
Operational Flexibility *(S)* (D_{16})	Capability to use other means of satisfying unexpected requirements than originally foreseen.	
	N (f_{58})	None
	O (f_{59})	Outsourcing
	F (f_{60})	Fulfillment Flexibility
	R (f_{61})	Routing Flexibility
	U (f_{62})	Unspecified *(S)*
*Inventory Rationing (D_{17})	Order in which backlog is fulfilled.	
	N (f_{63})	None
	F (f_{64})	First-Come-First-Served
	M (f_{65})	Maximum Fulfillment *(S)*
	P (f_{66})	Prioritized Customers *(S)*
	O (f_{67})	Other *(S)*
*Replenishment Rationing *(S)* (D_{18})	Order in which replenished items are divided (echelon replenishment).	
	N (f_{68})	None
	D (f_{69})	Depends on current inventory position per installation
	I (f_{70})	Independent of current inventory position per installation
Performance specification:		
Performance Indicator *(S)* (D_{19})	Objective to be achieved as a result of selection of control policy and its parameters.	
	E (f_{71})	Equilibrium
	S (f_{72})	Meeting operational service requirements
	C (f_{73})	Minimization of costs
	M (f_{74})	Multi-Objective
	U (f_{75})	Unspecified
*Costs *(S)* (D_{20})	Costs present in the inventory system.	
	N (f_{76})	None
	H (f_{77})	Holding costs
	R (f_{78})	Replenishment costs
	B (f_{79})	Backorder costs
*Service Level *(S)* (D_{21})	Performance measures used in the inventory system.	
	N (f_{80})	None
	A (f_{81})	The α-SL / ready rate
	B (f_{82})	The β-SL / fill rate
	G (f_{83})	The γ-SL
Scientific Aspects:		
Methodology (D_{22})	Techniques applied to achieve the results.	
	A (f_{84})	Approximative
	C (f_{85})	Computational experiments
	E (f_{86})	Exact
	F (f_{87})	Field study
	S (f_{88})	Simulation
Research Goal (D_{23})	Goal of the investigations.	
	C (f_{89})	Comparison
	F (f_{90})	Formulae
	O (f_{91})	Optimization
	P (f_{92})	Performance Evaluation

2.1.1 Classification string

We now have defined all different dimensions of a multi-echelon inventory system. In order to make sure we can easily compare different classifications, we need something more structured than the description of the features. Therefore, the classification strings of Van Santen (2019) and De Kok et al. (2018) are introduced. The original string of De Kok et al. (2018) has the following structure:

```
<No. of Echelons>, <Structure>, <Time>, <Information> | <Capacity>, <Delay> |
<Demand>, <Customer> | <Policy>, <Lot-size>, <Flexibility> |
<Performance Indicator> || <Methodology>, <Research Goal>
```

This string is divided into two parts, separated by a ||. The first part contains the dimensions related to the inventory system and consists of five sections, separated by a |. The second part contains information about the methodology of the paper. The string will be filled with the active feature(s) of the dimensions. An inventory system can have multiple active features per dimension. These values will then all be used in the string, without being separated by a comma. An example of the classification string is shown below:

```
n,G,D,G|F,G|G,B|b,F,R|C||CS,O
```

This string only contains the dimensions introduced by De Kok et al. (2018). The dimensions that are added by Van Santen (2019) are described in a separate string. For the ease of comparison with literature and to use a widely accepted classification structure, the typology of De Kok et al. (2018) is unaltered. The second string with the dimensions of Van Santen (2019) is structured as follows:

```
<No. of Products> | <> | <Intermediate Demand>, <Fulfillment>, <Substitution> |
<Review Period>, <Inventory Rationing>, <Replenishment Rationing> |
<Costs>, <Service Level>
```

To distinguish the two strings, we refer to them as T1 and T2. Therefore, an example of the final classification structure that Van Santen (2019) introduces is shown below:

```
T1: n,G,D,G|F,G|G,B|b,F,R|C||CS,O
T2: 2| |D,P,N|C,F,N|HR,B
```

2.2 Classification of CBC

In this section, we elaborate on the inventory system of CBC and classify it with the introduced notation. Each subsection covers the corresponding section of the classification method. At the end of this section, we state all the features in a classification string. The information about the supply chain of CBC is gathered from Van Santen (2019) and consultants of ORTEC that worked on the specific case.

2.2.1 Network specification

As mentioned in Section 1.3, the CardBoard Company is a multinational company that produces paper-based packaging. Their supply chain in Europe consists of *two echelons*. The first tier has four paper mills, which produce paper and are located in Keulen (DE), Linz (AT), Granada (ES) and Umeå (SE). The second tier produces cardboard in five corrugated plants, located in Aken (DE), Erftstadt (DE), Göttingen (DE), Valencia (ES) and Malaga (ES). Not all corrugated plants can be supplied by every paper mill, but it can always be supplied by at least two. The paper mills, on the other hand, can also always supply at least more than one corrugated plant. Because of these multiple connections, this supply chain is a *General Network*. Figure 2.1 shows the multi-echelon supply chain and its connections between

the different stock points.

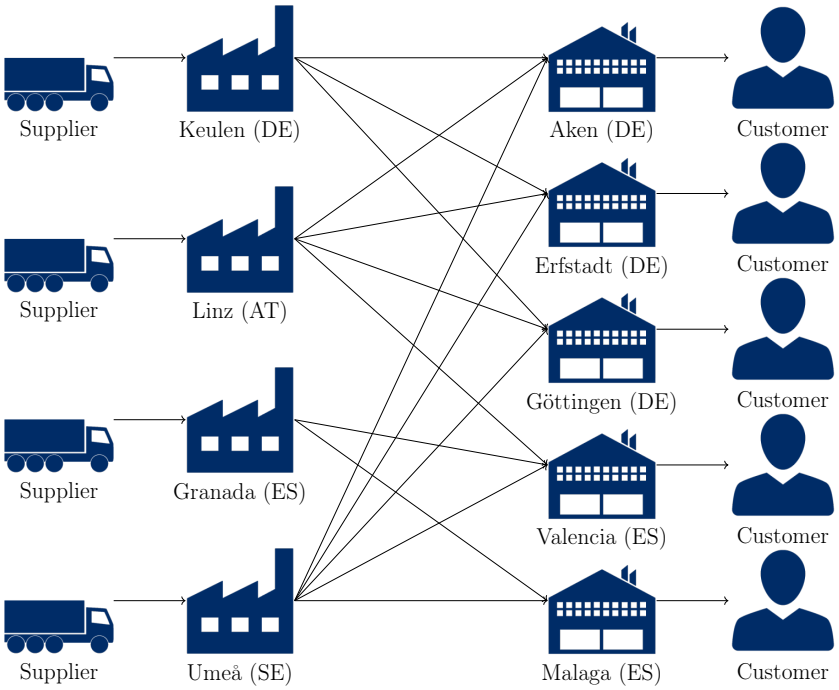


Figure 2.1: *An overview of CBC's supply chain*

In order to model the situation of CBC, we use a simulation in which the time is *discrete*. This means that each event occurs at a particular instant in time and marks a change of state in the system. Between consecutive events, no change in the system is assumed to occur (Robinson, 2004). The information level is *global*, because the CBC has a Supply Chain department that communicates with all the installations.

In this case, we only consider the products that CBC can produce in the corrugated plants in scope. Hence, the products in our model is the paper that is used to make the cardboards. These cardboards can differ in color, weight and size. In total, 281 unique product types can be produced by these corrugated plants. However, not every plant can produce every product type and there are types that can be produced in multiple corrugated plants. This results in a total of *415 product type - corrugated plant combinations*.

2.2.2 Resource specification

Every stock point in the supply chain has a *bounded capacity*. In this case, the capacity is known for every location and expressed in *kg*. This measure is used because the products are stored as paper rolls that can differ in weight and width. The weight generally increases linearly with the amount of paper. Because there may be situations where this relation is not (exactly) linear, we take a margin of 5% of the capacity per location. With this, we make sure that the solution will be feasible regarding the capacity and we can also account for capacity situations that are out of scope, for example, the size of the pallets. In the real situation of CBC, the capacity of several stock points can be expanded by the use of external warehouses, however, we consider these warehouses out of scope for our case.

A lead time is considered for transport between the paper mills and corrugated plants. We use a *constant* transportation time, but it will be different for every connection between these locations. This constant is determined by taking the average of the observed lead times for every connection. These observed

lead times did not vary more than 30 minutes on lengths of multiple hours. Therefore, we assume the variation in transportation time is negligible.

2.2.3 Market specification

For every corrugated plant, the historical demand is stored in the ERP system. The demand data is stored on an order level per calendar day. Hence, we can not distinguish the original sequence in which the orders arrived on the same day. The data we have is limited to one year. When we were to use this data in our model, we are likely to overfit. This happens when the model corresponds too closely to a limited set of data points and may fail to correctly predict the future observations (Kenton, 2019). In order to prevent overfitting the data in our model, we will fit distributions over this data. Because we will fit these distributions in a later stage, we will for now classify it as *general stochasticity*.

In the real-world situation of CBC, paper mills deliver not only to the corrugated plants, but also have other customers that can order directly at the paper mills. When stock points can receive orders from customers outside the considered supply chain, this is called *intermediate demand*.

When the demand exceeds the current inventory and the customers, therefore, can not be fulfilled immediately, orders are placed in the *backlog*. This means that these orders will be fulfilled whenever the location is replenished. In this case, it is also possible that customers are *partially fulfilled*, which happens whenever the replenishment was not enough to fulfill all the orders completely. The remaining items in this order are then backlogged. To make sure this partial fulfillment does not contain small orders and the transport costs will be too high, the locations consider a Minimum Order Quantity (MOQ), which holds for every order. In practice, whenever a product type is not available, the customer can also choose for *substitution* of the product, meaning that the customer can choose for an other, mostly more expensive, product to be delivered instead. In this case, the substitution product type can have a higher grade or grammage, but this differs for every product type. When a substitution product is chosen, demand for the original product is lost.

2.2.4 Control specification

In the current situation, the inventory of CBC is controlled by the Supply Chain department. They use an ERP system that is configured with a *(s,S) policy*. In this policy, s is the safety stock and S is the order-up-to-level. When ordering the replenishment for the stock points, the ERP system will give a recommendation, but the Supply Chain department finally decides on the order size and time. This ERP system checks the inventory at the end of every day; hence the review period is *periodically*.

Orders can be placed upstream for every kilogram. It is not possible to order in grams, hence, the order quantity will always be an integer. Therefore, the fixed order quantity will be set to one. Next to that, we also have to take the earlier mentioned Minimum Order Quantity into account here. For that reason, the constraint for lot-sizing is classified as *other*.

It could also be possible that a product type is not on stock on a certain corrugated plant, but is available at another plant. In this case, the product can be delivered from one corrugated plant to the other, which is described as *routing flexibility*. These connections, however, do only exist between a few plants that are closely located at each other.

Whenever two or more orders can only be partially fulfilled, an inventory rationing has to be realized, such as First-Come-First-Served. CBC uses an *other* method; the remaining stock is divided according to the relative amount of the order. For example, when two orders of 4 kg and 12 kg come in, the stock will be divided by 25% and 75% respectively.

Replenishment rationing needs to be applied when orders are placed at an echelon level. These orders then have to be divided upon the stock points (Van Der Heijden, Diks, & De Kok, 1997). In the case of CBC, there is *no replenishment rationing* because orders are placed on an installation level.

2.2.5 Performance specification

CBC wants to optimize its inventory by *minimizing the costs*, which, in this case, are the *holding costs and backorder costs*. While minimizing the costs, CBC also wants to achieve an average *fill rate* of 98% for the customers of its corrugated plants.

The real holding and backorder costs are unknown for the case of CBC. Because, we need costs in order to optimize the inventory of a supply chain, we have to estimate these costs. Based on the paper of Kunnumkal and Topaloglu (2011), we define holding costs of \$0.6 per kg for the paper mills and \$1 per kg for the corrugated plants. If items are back ordered at the corrugated plants, we denote costs of \$19 per kg. We do not take replenishment costs into account. We will decide for a price per kilogram, because all orders are placed per kilo. We will not differentiate between different products.

2.2.6 Scientific aspects

The dimensions *Methodology* and *Research Goal* do not involve the current situation of CBC, but are impacted by the methods we choose in this research. We will first model a *simulation* to reflect the inventory system of CBC. After that, we minimize the total inventory using a reinforcement learning method, which is an *optimization* goal. Reinforcement learning is a broad term which will be further explained in Chapter 3. When we should classify it according to the existing features, we could say that reinforcement learning uses *computational experiments* that *approximate* the optimal inventory policies.

2.2.7 Classification string

We now have discussed all dimensions that are relevant for an inventory system. The corresponding classification strings of the network of CBC are stated below. We use these strings to find similar situations in literature and to easily spot the important differences. This classification also makes it easier to see what kind of assumptions we can make when we want to simplify our situation.

```
T1: 2,G,D,G|F,C|C,B|s,O,R|SC|ACS,O
T2: n||M,P,I|P,O,N|N,B
```

2.3 Current method

As mentioned in the previous section, CBC has a supply chain department that handles the inventory. They use an ERP system that is configured with an (s,S) policy, but the planners make the final decision. The planners also decide on specific situations, such as the substitution of products.

At the moment, CBC wants to achieve a 98% fill rate for their customers. In order to realize this, CBC also required a 98% fill rate for the paper mills to the corrugated plants. ORTEC already showed that it is not necessary to require this fill rate at upstream stock points. Since it is not an end customer, it does not add direct value to your customers. Next to that, a high service level upstream does not necessarily lead to a significantly better service level downstream.

Unfortunately, the historic inventory levels were not available in the ERP system. Therefore, we have reconstructed these inventory levels, by determining the order-up-to level for a single product, for every stock point, in such a way that, for every connection, a 98% fill rate is achieved. This is in line with the

current method of CBC. Please note that these inventory levels are reconstructed using the simulation that is defined in Chapter 6, and, therefore, do not fully reflect the real-life situation of CBC. We measure the fill rate by running the simulation for 50 time periods and replicate this simulation 500 times. Figure 2.2 shows the reconstructed order-up-to levels based on the current policy of CBC. With these order-up-to levels, CBC yields the average total costs of 10467 over 50 time periods.

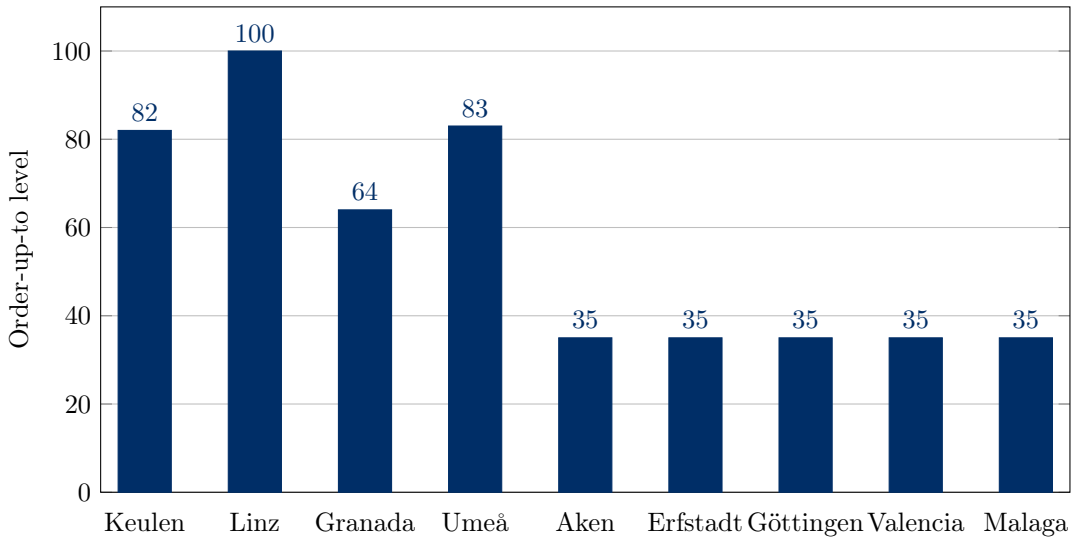


Figure 2.2: *Order-up-to levels for the stock points of CBC*

For this research, it is important to note that the stock points of CBC are production facilities. Because we only focus on the inventory, we only look at the finished paper rolls. Therefore, the scope covers the products after production in the paper mills and before production in the corrugated plants. As a result, the demand for corrugated plants is created by the internal demand of the production schedule. Also, the suppliers of the paper rolls in the paper mills are the production facilities of the mills. Because these production facilities are out of scope, we assume an unlimited production capacity.

CBC is interested in how their inventory can be reduced while achieving the desired fill rate of 98% for their customers. In order to achieve this, the supply chain planners are interested in a fitting inventory policy.

2.4 Conclusion

In this chapter, we provided an analysis of the current inventory system of the CardBoard Company, in order to answer our first set of research questions:

1. **What is the current situation at the CardBoard Company?**
   (a) *How can we describe relevant aspects of multi-echelon inventory systems?*
   (b) *What are the specific characteristics of CBC?*
   (c) *What is the current performance for inventory management at CBC?*

In Section 2.1, we introduced the classification method of De Kok et al. (2018) and Van Santen (2019). This classification is designed to capture all the relevant aspects of a multi-echelon inventory system. While the classification of De Kok et al. (2018) mainly focused on the relevant aspects that are described in cases in literature, the goal of Van Santen (2019) was to extend this classification in order to let it reflect the complexity of the real-world.

We use the classification of Van Santen (2019) to describe the inventory system of CBC in Section 2.2.

We were able to successfully determine a feature for every dimension in the classification, that resulted in the following classification strings:

```
T1: 2,G,D,G|F,C|C,B|s,O,R|SC|ACS,O
T2: n||M,P,I|P,O,N|N,B
```

With this classification, we can now easily compare the situation of CBC with inventory systems in literature. The classification also makes it easier to see what kind of assumptions we can make when we want to simplify our situation.

In Section 2.3, we focused on the current performance of CBC. While the original inventory levels were not available in the ERP system, we were able to reconstruct these levels using the inventory policy that CBC currently uses. In this policy, CBC aims to achieve a 98% fill rate to their customers, but also wants to achieve this fill rate for connections between the paper mills and corrugated plants. In order to reduce the inventory levels, we will introduce a reinforcement learning method which aims to find an improved inventory policy, to minimize the total inventory costs.

In the next chapter, we will do research on other multi-echelon inventory systems and the application of reinforcement learning in these systems.

3. Literature Review

In this chapter, we will discuss the findings of the literature review. We start with an explanation of reinforcement learning and its elements in Section 3.1. Section 3.2 covers the combination of deep learning with reinforcement learning. In Section 3.3, we discuss existing methods for inventory management in a multi-echelon supply chain. We address the current types of reinforcement learning in supply chains and their requirements in Section 3.4. In Section 3.5, we cover the practical usage of reinforcement learning. The conclusion of this literature research is stated in Section 3.6.

3.1 Reinforcement Learning

We briefly introduced reinforcement learning in Section 1.2. In this section, we will further elaborate on the structure of reinforcement learning. Thereafter, we discuss how approximate dynamic programming is related to reinforcement learning, and subsequently, we discuss a widely used method, temporal difference learning, accompanied by the two most used algorithms.

3.1.1 Elements of Reinforcement Learning

Reinforcement learning is about learning what to do to maximize a reward. In reinforcement learning, other than in most forms of machine learning, the learner is not explicitly told which action to perform in each state (Sutton & Barto, 2018). At each time step, the agent observes the current state s_t and chooses an action a_t to take. The agent selects the action by trial-and-error (exploration) and based on its knowledge of the environment (exploitation) (Giannoccaro & Pontrandolfo, 2002). After taking the action, the agent is given a reward r_{t+1} and evolves in the new state s_{t+1}. Using this information, the agent updates its knowledge of the environment and selects the next action. To maximize the reward, the trade-off between exploration and exploitation is very important. The agent has to explore new state-action pairs in order to find the value of the corresponding reward. Full exploration ensures that all actions are taken in all states. Ideally, with the values that are learned in this phase, a trend can be found, such that it is clear which is the best action in each state. The vector that maps every state into the associated optimal action represents the learned optimal policy (Giannoccaro & Pontrandolfo, 2002). Most of the methods start with full exploration and slowly decrease the chance of exploration, hence increasing the chance for exploitation.

3.1.2 Markov Decision Process

Every state that the agent enters is a direct consequence of the previous state and the chosen action. Each step and the order in which they are taken hold information about the current state and have an effect on which action the agent should choose next (Zychlinski, 2019). However, with large problems, it becomes impossible to store and process all the information. To tackle this, we assume all states are Markov states. This means that any state solely depends on the state that came directly before it and the chosen action. When a reinforcement learning system satisfies this Markov property, it is called a Markov Decision Process (MDP) (Sutton & Barto, 2018). When we assume that both the set of all possible states is finite, as well as the set of all possible actions in each state, we have a finite MDP. The finite MDP is mostly used in reinforcement learning, because this way we can assure the one-step dynamics. The probability definition of a finite MDP is:

$$P(s', r|s, a) = P\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{3.1}$$

and denotes the probability of any possible next state s', given a state-action pair. Figure 3.1 shows a transition graph, which is a visualization of a Markov Decision Process. This example concerns a robot

that can be in two states depending on the battery level: either low or high, which are given in the large circles. The black dots denote the state-action pairs. On each arrow, the transition probability and reward are shown. Note that the transition probabilities labeling the arrows leaving an action node always sum to one (Sutton & Barto, 2018).

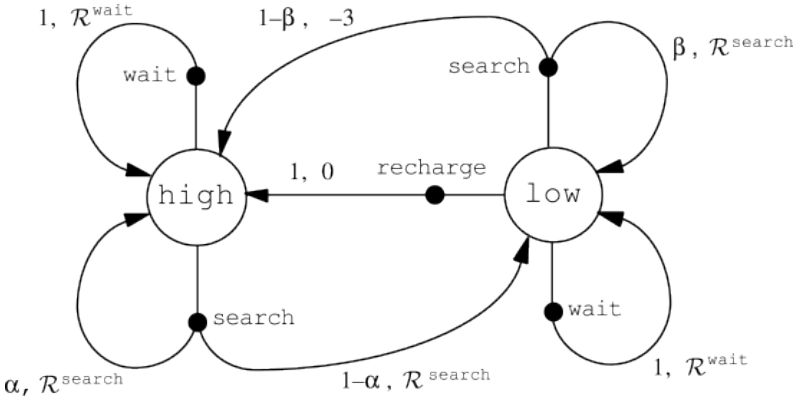


Figure 3.1: *Transition graph of a finite MDP (Sutton & Barto, 2018).*

3.1.3 Bellman Equation

When using reinforcement learning, it is important to choose the best action for every state. The Bellman equation is used to learn which action is the best action. The Bellman equation is a general formula that is based on the fact that the value of your starting point is the reward you expect to get from being in that state, plus the value of the state you will enter. The Bellman equation of a deterministic environment is denoted as follows (Bellman, 1957):

$$V(s) = \max_a(R(s,a) + \gamma V(s`)) \tag{3.2}$$

In this equation, $R(s,a)$ is the reward value of a certain action a in state s. $\gamma V(s')$ denotes the discounted value of its next state. With the Bellman equation, all possible actions in a state are considered and the one with the maximum value will be chosen. The Bellman equation is one of the core principles of reinforcement learning and used, most of the times in adapted versions, in the reinforcement learning methods.

3.1.4 Approximate Dynamic Programming

Simple MDPs can be solved by evaluating all possible policies. Obviously, this will become computationally intractable for larger and more complicated MDPs. Another, smarter, way to solve MDPs is Dynamic Programming (DP). Dynamic Programming solves complex MDPs by breaking them into smaller subproblems (Mes & Rivera, 2017). The optimal policy for the MDP is one that provides the optimal solution to all sub-problems of the MDP (Bellman, 1957). DP updates the estimates of the values of the states based on the estimates of the values of the successive states, or update the estimates based on past estimates. Dynamic Programming requires a perfect model of the environment, which must be equivalent to an MDP. For this reason, and because of its high computation costs, the DP algorithms are of little use in reinforcement learning. Yet, they represent the theoretical framework of reinforcement learning, because both methods try to achieve the same goal. The advantage of RL is that it can do it with lower computation costs and without the assumptions of a perfect model (Sutton & Barto, 2018).

Approximate Dynamic Programming (ADP) is a modeling framework based on an MDP model. ADP is an umbrella term for a broad spectrum of methods to approximate the optimal solution of an MDP, just

like reinforcement learning. It typically combines optimization with simulation. Because Approximate Dynamic Programming and reinforcement learning are two closely related paradigms for solving sequential decision-making problems, we will both discuss them in the section about reinforcement learning in inventory management.

3.1.5 Temporal Difference Learning

With a Markov Decision Process, we have a complete model of the environment, including transition probabilities. However, RL can also be used to obtain an optimal policy when we do not have such a model (Kaelbling et al., 1996). There are two approaches that convert the experience of interacting with the model into information that can be used for determining an optimal policy. In these approaches, it is not necessary to know the transition probabilities. One approach is to derive a controller directly from experience without creating a model and is called the model-free approach. The other approach, which is called the model-based approach, is to learn a model from experience and derive a model from that (Kaelbling et al., 1996).

In reinforcement learning, it is important that the agent knows how to assign temporal credit: an action might have an immediate positive credit, but large negative effects in the future (Sutton, 1984). A way to handle this is to assign rewards to every action, according to the end result, at the end of an episode. This method requires a large amount of memory and is therefore not always suitable. In this case, we can use temporal difference methods, which estimates the value of a state-action pair from an immediate reward and the expected reward from the next state (Sutton & Barto, 2018). This way, the estimates are updated more frequently than only at the end of an episode, but still takes the future rewards into account. Temporal difference learning is often described as TD(λ) or TD(0). In case of the latter, or when lambda is set to 0, the algorithm will be a one-step algorithm, so the estimate takes the value of one next state into account. If λ is set to 1, the formulation will be a more general way of implementing a Monte Carlo approach. Any λ value between 0 and 1 will form a complex n-step return algorithm (Çimen & Kirkbride, 2013). Two algorithms of temporal difference that are often used are Q-learning and SARSA, which are covered in the next subsections.

Q-Learning

Q-learning is one of the most used temporal difference algorithms because it is easy to implement and it seems to be the most effective model-free algorithm for learning delayed rewards (Kaelbling et al., 1996). It is an off-policy algorithm, which means that the Q-learning function learns from actions that are outside the current policy, such as random actions, hence a policy is not needed (Violante, 2019). The Q-values of a state-action pair are based on the Bellman equation. Therefore, they are defined as follows:

$$Q^*(s,a) = R(s,a) + \gamma \max_a Q^*(s',a) \tag{3.3}$$

Hence, the value of taking action a in state s is defined as the expected reward starting from this state s and taking the action a. Q-learning is a tabular algorithm, meaning that it stores all the information in a table. For every state-action pair, it learns an estimate $Q(s,a)$ of the optimal state-action value $Q^*(s,a)$, according to the following value function (Watkins, 1989):

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a Q(s',a) - Q(s,a)] \tag{3.4}$$

The γ in Formula 3.4 determines the discount factor, that concerns the importance of future rewards. A factor of 0 will make the agent myopic by only considering current rewards r. On the other hand, a value of 1 will make it strive for a high rewards on the long-term.

The α denotes the learning rate and has a value between 0 and 1. A value of 0 would result in full exploitation, while a value of 1 results a full exploration. It has been shown that, if the learning rate is decayed at an appropriate rate, the estimates converge to the optimal values for all possible pairs (Jaakkola, Jordan, & Singh, 1994; Littman, 2015):

$$Q(s,a) \rightarrow Q^*(s,a) \tag{3.5}$$

Although this exploration-exploitation must be addressed using the α, the details of the exploration strategy will not affect the convergence of the learning algorithm. Therefore, Q-learning is exploration insensitive, meaning that the Q values will converge to the optimal values, independent of how the agent behaves while the data is being collected (as long as all state-action pairs are tried often enough). This property makes Q-learning very popular (Kaelbling et al., 1996). However, because state-action pairs have to be visited often, this also results in the fact that Q-learning is a slow algorithm. Next to that, being a tabular algorithm also makes it memory intensive, as all states have to be stored in a table.

The pseudo-code of the Q-learning algorithm can be found in Algorithm 1 (Sutton & Barto, 2018).

***Algorithm 1:** Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$*

Algorithm parameters: step size $\alpha \in (0,1]$, small $\epsilon > 0$;
Initialize $Q(s,a)$, for all $s \in S^+$, $a \in A(s)$, arbitrarily except that $Q(terminal,\cdot) = 0$;
foreach *episode* **do**
    Initialize S;
    foreach *step of episode* **do**
        Choose A from S using policy derived from Q (e.g., ϵ-greedy);
        Take action A, observe R, S';
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \cdot \max_{a'} Q(S',a) - Q(S,A)]$;
        $S \leftarrow S'$;
    Until S is terminal;

SARSA

Another temporal difference algorithm is SARSA. SARSA is an acronym for State-Action-Reward-State-Action $(S_t, A_t, R_t, S_{t+1}, A_{t+1})$. Opposed to Q-Learning, SARSA is an on-policy algorithm, meaning that it updates the policy directly on taking the action. When making moves, they follow a control policy and use it to update the Q-Values, according to the following formula (Sutton & Barto, 2018):

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)] \tag{3.6}$$

SARSA also uses a table to store every state-action pair and estimates q_π according to the behavior policy π and at the same time changes π toward greediness with respect to q_π. In other words, it estimates the return for state-action pairs assuming the current policy continues to be followed and in the meantime becomes less exploratory and more exploitative. The pseudo-code of the algorithm can be found in Algorithm 2.

Q-Learning versus SARSA

We can think of SARSA as a more general version of Q-learning, the algorithms are very similar, as we can see in Algorithm 1 and 2. In practice, we can modify a Q-learning implementation to SARSA by changing the update method for the Q-values and by choosing the current action A using the same policy

Algorithm 2: SARSA (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$;
Initialize $Q(s, a)$, for all $s \in S^+$, $a \in A(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$;
foreach *episode* **do**
    Initialize S;
    Choose A from S using policy derived from Q (e.g., ϵ-greedy);
    foreach *step of episode* **do**
        Take action A, observe R, S';
        Choose A' from S' using policy derived from Q (e.g., ϵ-greedy);
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \cdot Q(S', A') - Q(S, A)]$;
        $S \leftarrow S'$;$A \leftarrow A'$;
    Until S is terminal;

in every episode. Inside every step, we then choose A' instead of A. There can, however, be a significant difference in performance. To illustrate this, we use a famous example of (Sutton & Barto, 2018); the cliff-walking problem.

In this cliff-walking problem, there is a penalty of -1 for each step that the agent takes and a penalty of -100 for falling off the cliff. The optimal path is, therefore, to walk exactly along the edge of the cliff and reach the reward in the least steps as possible. This maximizes its reward as long as it does not fall into the cliff at any point.

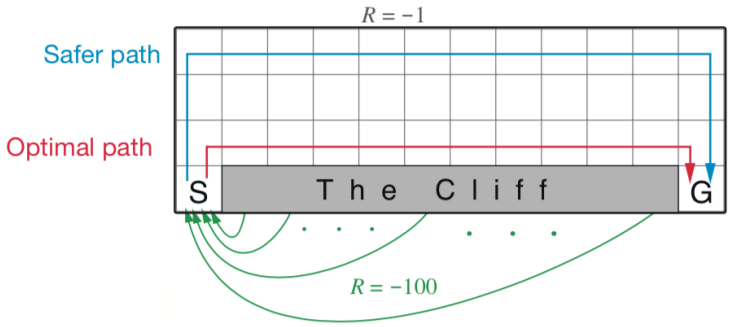


Figure 3.2: *The cliff-walking problem (Sutton & Barto, 2018).*

Figure 3.2 shows two different paths. Q-learning takes the optimal path, while SARSA takes the safe path. The result is that, with an exploration-based policy there is a risk that at any point, a Q-learning agent will fall off the cliff as a result of choosing exploration.

SARSA looks ahead to the next action to see what the agent will actually do at the next step and updates the Q-value of its current state-action pair accordingly. For this reason, it learns that the agent might fall into the cliff and that this would lead to a large negative reward, so it lowers the Q-values of those state-action pairs accordingly. The result is that Q-learning assumes that the agent is following the best possible policy without attempting to resolve what that policy actually is, while SARSA takes into account the agent's actual policy.

When the agent's policy is simply the greedy one, Q-learning and SARSA will produce the same results. In practice, most implementations do not use a simple greedy strategy and will instead choose something such as e-greedy, where some of the actions are chosen at random.

In many problems, SARSA will perform better than Q-learning, especially when there is a good chance that the agent will choose to take a random sub-optimal action in the next step, as we can see in the cliff-walking problem. In this case, Q-learning's assumption that the agent is following the optimal policy may be far enough from true that SARSA will converge faster and with fewer errors (Habib, 2019).

3.2 Deep Reinforcement Learning

In the reinforcement learning methods that we discussed until now, we assumed that we can explore all states and represent everything we have learned about each state in a table. However, when we want to use these methods to solve large, realistic problems, the amount of states and actions is too large to store all the information in a table. Next to that, not only the storage is a limiting factor, as we also need a large amount of time to estimate each state-action pair. These limitations can be described as a generalization problem. To overcome these limitations, we need to use a method that can generalize the experience that it gained when training on a subset of state-action pairs to approximate a larger set. Value function approximation is such a method. With value function approximation, we can represent the Q-values by using a function instead of a table, for which we only need to store the parameters. This way, it requires less space and it can generalize visited states to unvisited states, so we do not need to visit every possible state. There are several ways to use value function approximation for reinforcement learning, but a method that is gaining popularity is the use of deep learning (DL). Deep learning means that the learning method uses one or multiple neural networks to estimate the value function. In a recent paper, Mnih et al. (2015) applied deep learning as a value function approximator and gained impressive results. Deep learning was combined with reinforcement learning, because they have a common goal of creating general-purpose artificial intelligence systems that can interact with and learn from its world (Arulkumaran, Deisenroth, Brundage, & Bharath, 2017). While reinforcement learning provides a general-purpose framework for decision-making, deep learning provides the general-purpose framework for representation learning. With the use of minimal domain knowledge and a given objective, it can learn the best representation directly from raw inputs (Sutton & Barto, 2018).

3.2.1 Neural Networks

A neural network consists of multiple layers of nonlinear processing units. This network takes input and transforms it to outputs. An example of a neural network with two hidden layers and one output is illustrated in Figure 3.3.

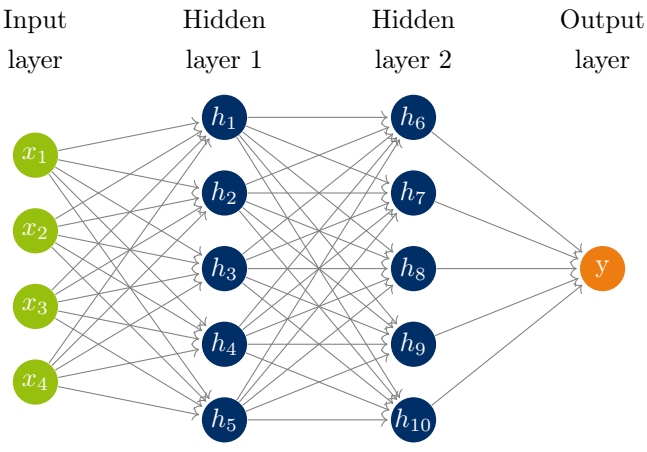


Figure 3.3: *A visualization of a neural network.*

Every layer of the neural network contains a number of neurons. A neuron is a computational unit with

a weight parameter and is visualized in Figure 3.4. This neuron takes input from the previous layer and multiplies this by the weight of the connection. These weighted inputs are added together with a bias. At last, the sum is passed through an activation function. In the case of the example of Figure 3.4, this would result in the following activation function:

$$y = f(x_1 * w_1 + x_2 * w_2 + b) \tag{3.7}$$

The activation function is being used to transform an unbounded input to an output in a predictable form and is typically nonlinear. There are multiple kinds of activation functions, such as the Sigmoid function. This function has an output in the range of 0 and 1, so it compresses the values. The outcome of this activation function is then passed to the next layer. The output layer is computed the same way as the hidden layer, except it provides the final value. The correct output is then calculated using a loss function on the output layer, for which the mean squared error or the log-likelihood function are often used (Russell & Norvig, 2009). The lower this loss is, the better the prediction.

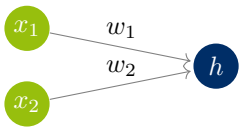


Figure 3.4: *A neuron inside a neural network.*

While training a neural network, input is given to the network with a known output. With this input, the weight and biases of the network can be adjusted in such a way that it leads to the correct output. This is often done by using stochastic gradient descent, which can calculate exactly how much weight and bias is needed for every connection and neuron. Once a certain loss value or accuracy of the method has been achieved, new data can be added to the network. The neural network can then calculate the estimated output of a given input. When combining deep learning with reinforcement learning, the input is the defined state. With this input layer, the network can estimate which action has to be performed. The output layer corresponds to the action space. This can either be by returning a certain estimate and standard deviation per action or by returning the probability for all actions. Gained rewards or penalties can be used to update the network. If a penalty is given, the weights will be adjusted in such a way that, when the agent ends up in the same space, another action will be chosen.

3.2.2 Algorithms

Just like in reinforcement learning, several algorithms are developed to work with deep reinforcement learning. In the next subsections, we discuss popular deep reinforcement learning algorithms.

Deep Q-Network (DQN)

Mnih et al. (2015) introduced a new value function based DRL method: The Deep Q-Network. In this paper, a single architecture is used to learn to play a wide variety of Atari 2600 games. DQN outperformed competing methods in nearly all games. It is the first reinforcement learning algorithm that could directly use the images of the game as input and could be applied to numerous games. While Q-learning uses a table and stores Q-values for all the state-action combinations, DQN uses a neural network to estimate the performance of an action. In DQN, Q-values are calculated for a specific state, instead of for a state-action pair in Q-learning. This difference is visualized in Figure 3.5. The concept of using states instead of state-action pairs is also often used in ADP (Powell, 2011). In ADP, this is used the post-decision state variable, which captures the state of the system directly after a decision is made, but before any new information has arrived. The advantage of using the post-decision state variable is that it is not necessary to evaluate all possible outcomes for every action, but only for the different states (Mes & Rivera, 2017).

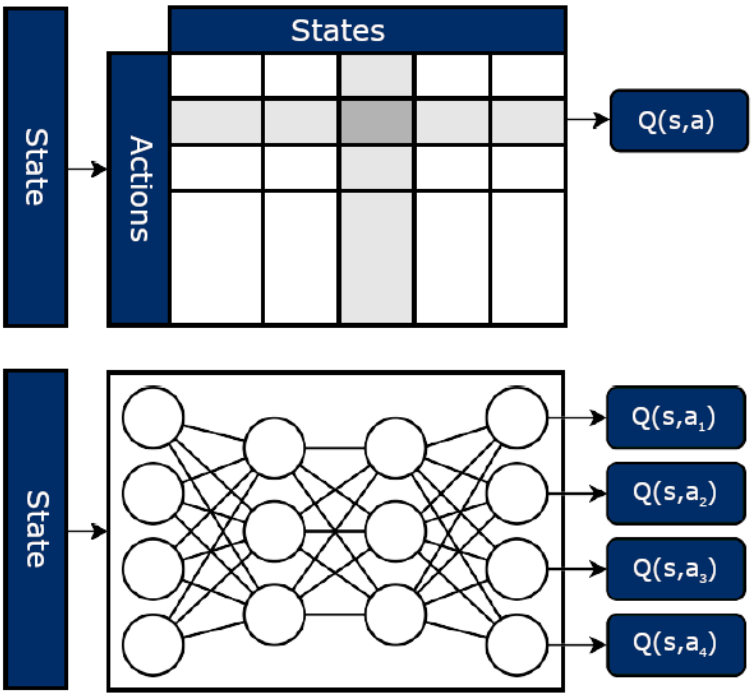


Figure 3.5: *A visualization of the differences between a Q-table and a Q-network. Adapted from Gemmink (2019).*

The value function approximator for DQN is shown in Equation 3.8 (Mnih et al., 2015). Note that this equation shows both the state and action, because we still want to approximate the Q-values using both the state and action.

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi] \tag{3.8}$$

This equation defines the best sum of rewards (r_t, r_{t+x}) that are discounted over time by discount factor γ, achievable by a behavior policy $\pi = P(a|s)$, after taking action a in state s (Mnih et al., 2015). While other value function approximations in reinforcement learning are known to become unstable, Mnih et al. (2015) solves this instability issue with two new ideas: experience replay and iterative updates of the target network. With experience replay, the agent's experiences of every time step are stored in a data set and when a q-learning update iteration is executed, a sample of these experiences is drawn at random. With this method, correlations in observation sequences are removed. The target network is only updated after a number of iterations. A requirement for experience replay is that the laws of the environment are not changed in such a way that past experiences become irrelevant (Lin, 1991). With these ideas, Mnih et al. (2015) is able to successfully parameterize the approximate value function.

Actor-Critic (AC)

The Actor-critic algorithm was first introduced by Lillicrap et al. (2016) and is a combination of value function based methods (such as Q-learning and DQN) and policy based methods. In Actor-critic methods, the actor acts as a policy and updates the policy distribution in the direction suggested by the critic. The critic estimates the value function. The estimation is based on a value function approximator of the returns of all actions of the environment. This algorithm gives smaller gradients, which results in more stable updates with less variance. The architecture of the Actor-Critic methods is illustrated in Figure 3.6. In Mnih et al. (2016), the actor-critic algorithm is expanded to the Asynchronous Advantage Actor-

Critic (A3C) algorithm. It is asynchronous, because it executes multiple agents on multiple instances of the environment in parallel, with the same deep neural network. Instead of experience replay, updating is done by the experience of the different actors. This way, it reduces the chance of getting stuck in a local optimum.

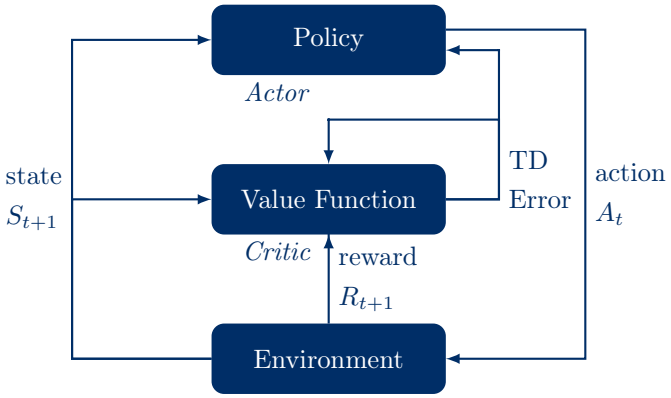


Figure 3.6: *The Actor-Critic architecture (Sutton & Barto, 2018).*

Proximal Policy Optimization (PPO)

Currently, one of the most successful DRL methods is the proximal policy optimization (PPO) method, based on the Actor-Critic architecture, and was introduced by Schulman et al. (2017), from OpenAI. Policy search methods usually have convergence problems, which are addressed by the natural policy gradient. This natural policy gradient involves a second-order derivative matrix, which makes this method not scalable for large problems. PPO uses a slightly different approach. It can use the gradient descent method to optimize the objective. This way, the objective is updated gradually, to ensure the new policy will not move too far away from the old policy. This makes the algorithm less sensitive too outliers in its observations. Schulman et al. (2017) compares its PPO method with the DQN of Mnih et al. (2015) and outperforms this method in several cases. PPO can make efficient samples and is computationally less intensive.

3.3 Multi-echelon Inventory Management

The term multi-echelon is used for supply chains when an item moves through more than one stage before reaching the final customer (Ganeshan, 1999; Rau, Wu, & Wee, 2003). In other words, a product is moving through different locations in this supply chain. Because of the multiple locations, managing inventory in a multi-echelon supply chain is considerably more difficult than managing it in a single-echelon one (Gumus, Guneri, & Ulengin, 2010). Because of its complexity, research on multi-echelon inventory models is still gaining importance. In the last decades, the research is focusing more on integrated control of the supply chain, mainly due to modern information technology (Gumus & Guneri, 2009; Kalchschmidt, Zotteri, & Verganti, 2003; Rau et al., 2003). Literature reviews conducted on multi-echelon inventory management describe several methods to optimize the inventory levels. According to Gumus and Guneri (2007), mathematical models are mostly used, often in combination with simulation. After that, heuristics and Markov Decision Process are most used.

The aim of mathematical models is to solve the inventory problem to proven optimality. However, determining optimal policies is intractable, even for basic multi-echelon structures, due to the well-known curses of dimensionality (Bellman, 1957; De Kok et al., 2018). As a result, most mathematical models use extensive restrictions and assumptions, such as deterministic demand and constant lead times. Also,

mathematical models are usually complex and very case-specific. We are looking for a method that can be used as a benchmark in order to compare it with the results of our reinforcement learning methods. This method has to be easy to implement and will be used for various cases. For these reasons, mathematical models are not suitable as a benchmark for our reinforcement learning method.

For this research, we want to gain insight in the methods that can be useful for our reinforcement learning model as a benchmark (Heuristics) or as a building block (Markov Decision Processes).

3.3.1 Heuristics

Due to the intractability of mathematical models, heuristics are often used in the optimization of multi-echelon inventory management. Heuristics are used to find a near-optimal solution in a short time span and are often a set of general rules, which can, therefore, be used on various problems. Several papers about reinforcement learning in multi-echelon inventory management use heuristics as a benchmark.

Van Roy, Bertsekas, Lee, and Tsitsiklis (1997) use a neuro-dynamic programming solution approach to solve their inventory problem. To benchmark their solution, they use a heuristic to optimize the order-up-to policy (i.e., base-stock policy). This heuristic has been the most commonly adopted approach in inventory optimization researches (Nahmias & Smith, 1993, 1994). For multi-echelon inventory systems, the base-stock policy, though not necessarily optimal, has the advantage of being simple to implement and close to the optimum (Rao, Scheller-Wolf, & Tayur, 2000).

Oroojlooyjadid, Nazari, Snyder, and Takáč (2017) also use a heuristic, introduced by Snyder (2018) to choose the base-stock levels. Gijsbrechts et al. (2019) define three scenarios to apply their reinforcement learning method to, dual-sourcing, lost sales, and multi-echelon. They use several heuristics as benchmark for the first two scenarios, such as Single Index, Dual-Index, and Capped-Dual-Index. They also use a Linear Programming based Approximate Dynamic Programming algorithm, introduced by Chen and Yang (2019), but this algorithm did not manage to develop policies that outperformed the above heuristics. For the multi-echelon scenario, they state that it is not possible to deploy the LP-ADP approach, as the structure of the value function is unknown and the K-dimensional demand makes the number of transition probabilities too large and the formulation of the constraints of the LP infeasible. Therefore, they also use a base-stock policy with constant base-stock levels as benchmark.

3.3.2 Markov Decision Process

Several papers use a Markov Decision Process (MDP) to model their supply chain. As we discussed in Section 3.1, MDPs are an important aspect of reinforcement learning. Almost all papers that use an MDP to solve their supply chain apply their method to a linear supply chain. In a linear supply chain, every stock point in this supply chain has only one predecessor and only one successor. Lambrecht, Muchstadt, and Luyten (1984) showed that the multi-echelon stochastic problem can be viewed as a Markov Decision Process.

Lambrecht, Luyten, and Vander Eecken (1985) use this finding to study the allocation of safety stocks and safety times of a two-echelon, single-item inventory system. This process heavily depends on the problem characteristics such as demand variability, cost structure, and lead time structure. Yet, they are able to determine near-optimal policies. Chen and Song (2001), Vercraene and Gayon (2013) and Iida (2001) use an MDP in a linear n-echelon supply chain. The main purpose of Iida (2001) is to show that near-myopic policies are acceptable for a multi-echelon inventory problem. The Markov Decision Process is also used in a divergent supply chain in the papers of Chen, Feng, and Simchi-Levi (2002) and Minner,

Diks, and De Kok (2003).

If we were to use an existing MDP as a building block for our reinforcement learning method, it must reflect a similar situation as the case of CBC, which has a *General* network structure. As discussed in Chapter 2, we consider a network structure to be general if this network contains at least one stock point that can be supplied by more than one upstream location, while also containing at least one stock point that can supply multiple downstream locations. To the best of our knowledge, Markov Decision Processes have not been applied to a *General* network structure. Therefore, we will resort to the literature of reinforcement learning to use and adapt one of their Markov Decision Processes.

3.4 Reinforcement Learning in Inventory Management

As we are interested in how we can apply reinforcement learning to the situation of the CardBoard Company, we look into the literature to see how reinforcement learning is used in other multi-echelon models. However, reinforcement learning in multi-echelon inventory management is still a broad concept where various multi-echelon systems and solution methods can be used. At the moment, quite some research on this subject has been done. We want to see which papers are close to the case of the CardBoard Company and what solution approaches are most used. This way, we can select the papers that fit closely to our research and, therefore, are most relevant. To gain insight into the different multi-echelon systems that are described in the literature and the solution approaches that are being used, we create a concept matrix. This matrix consists of elements of a multi-echelon inventory system that are relevant for our reinforcement learning method. These elements are extracted from the classification typology of Van Santen (2019). Table 3.1 shows the classification elements and the possible values of these elements. In this table, the first column states the classification elements, while the other two columns list the possible values of this classification element.

Table 3.1: *Classification elements and values*

Classification element	Values	
Network structure	Linear	Divergent
	Convergent	General
Products	Single	Multiple
Unsatisfied demand	Lost Sales	Backorders
	Hybrid	
Goal	Minimize Costs	Maximize Profit
	Target Service Level	
Uncertainties	Demand	Lead time
Horizon	Finite	Infinite
Algorithm	Various	
Approach	Reinforcement Learning (RL)	Deep Reinforcement Learning (DRL)
	Neuro-Dynamic Programming (NDP)	Approximate Dynamic Programming (ADP)

The first classification element, the network structure, has four different values: Linear, Divergent, Convergent, and General. As mentioned earlier, a network is linear when every stock point in this network has one predecessor and one successor. It is divergent if the stock points have one predecessor but multiple successors and convergent whenever the stock points have multiple predecessors and only one successor. In a general network, stock points have multiple predecessors ánd multiple successors.

The second element, products, defines the number of products that are taken into account in the solution approach of the paper. We make a difference between focusing on a single product and multiple products

at the same time.

The element unsatisfied demand denotes what happens whenever a stock point does not have enough stock to satisfy all demand. In some cases, this demand is lost, resulting in lost sales. Other cases use backorders, which means that the demand will be satisfied at a later point in time. In hybrid cases, there is a certain chance that the customer wants to backorder, otherwise the demand is lost.

The goal of the paper is the next element and can be to either minimize the costs, maximize the profit or to reach a target service level. If the goal is to reach a target service level, the method will try to minimize the difference between the realized service level and the target service level, both in a positive and a negative difference.

The next element is the uncertainties that are taken into account in the paper, which can be the demand and the lead time. For these uncertainties, a distribution is used and the values will be generated.

The next element is the time horizon, which can be either finite or infinite. In a finite horizon, the paper simulates a certain number of days or weeks. The outcome of the method will be a set of actions for every time period that is simulated. In an infinite horizon, the simulation runs for a very long period. The outcome is a set of parameters that can be used for a predefined inventory policy.

The possible values of the next element, algorithm, are various. This can be algorithms that are well known in reinforcement learning, but can also be new algorithms that are only used in a specific paper.

The last element is the approach that is used in the paper. This can be reinforcement learning, deep reinforcement learning, approximate dynamic programming, or neuro-dynamic programming. Neuro-dynamic programming and approximate dynamic programming are both synonyms of reinforcement learning (Bertsekas & Tsitsiklis, 1996).

The concept matrix is filled with papers that use reinforcement learning - or one of the earlier mentioned synonyms - to solve a multi-echelon inventory problem. There are 29 papers published with this subject that are relevant for this thesis. The paper is denoted in the first column of the concept matrix, after which the classification elements are stated. Sometimes, the paper did not clearly describe a certain feature, in that case, the classification element is left blank in the table. The last column of the concept matrix is reserved for additional notes on aspects that do not immediately fit in the structure of the concept matrix, but are interesting characteristics of the paper. The complete concept matrix is given in Table 3.2.

Table 3.2: *Usage of reinforcement learning in inventory management. (1/2)*

Paper	Network structure	Products	Unsatisfied demand	Goal	Uncertainties	Horizon	Algorithm	Approach	Additional Notes
[1]		Single, Multiple	Backorders	Minimize costs	Demand, Lead time	Finite	SMART	RL	Simulation optimization
[2]	Linear	Single	Backorders	Minimize costs	Demand, Lead time	Finite	Q-Learning	RL	
[3]	General	Multiple	Lost sales	Minimize costs	Demand	Infinite	TD(λ)	ADP	Production system
[4]	General	Multiple	Lost sales	Minimize costs	Demand	Infinite	Q-Learning TD(λ)	ADP	Production system
[5]	Divergent	Single	Lost sales	Minimize costs	Demand	Finite	Double-pass	ADP	Centralized and decentralized control
[6]	Linear	Single	Backorders	Maximize profit	Demand, Lead time	Infinite	SMART	RL	
[7]	Divergent	Single	Hybrid	Minimize costs	Demand	Infinite	A3C	DRL	Also applied its technique on two other, non multi-echelon problems
[8]	Divergent	Single	Lost sales	Target service level	Demand	Infinite	Case-based	RL	Fixed group of customers and competitive market
[9]	Divergent	Single	Lost sales	Target service level	Demand	Infinite	Action-Value	RL	Non-stationary demand, centralized and decentralized control, Adjust safety lead time
[10]	Linear	Single	Backorders	Minimize costs	Demand	Infinite	Asynchronous Action-Reward	RL	Adjust safety lead time
[11]	Linear	Single		Target service level	Demand		Distributed supply chain control algorithm	RL	Adjust safety lead time
[12]	Divergent	Single	Backorders	Minimize costs	Demand	Finite	Lagrangian relaxation	ADP	
[13]	Linear	Single	Backorders	Minimize costs	Demand	Infinite	Retrospective action reward	RL	
[14]	Divergent	Single	Lost sales	Target service level	Demand	Infinite	Case based myopic	RL	VMI
[15]	Divergent	Single	Lost sales	Target service level	Demand	Infinite		RL	Stationary and non-stationary demand
[16]	Linear	Single	Backorders	Minimize costs	Demand	Infinite	DQN	DRL	
[17]	Divergent	Single	Lost sales	Maximize profit	Demand	Infinite	Q-learning	RL	Single agent and non-cooperative multi agent, inventory sharing between retailers
[18]	Divergent	Single	Lost sales	Maximize profit	Demand	Infinite	Q-learning	RL	Single agent and non-cooperative multi agent, inventory sharing between retailers
[19]	Linear	Single	Backorders	Minimize costs	Demand	Infinite	Q-learning, Profit sharing	RL	

Table 3.2: *Usage of reinforcement learning in inventory management. (2/2)*

Paper	Network structure	Products	Unsatisfied demand	Goal	Uncertainties	Horizon	Algorithm	Approach	Additional Notes
[20]	Divergent	Single		Minimize costs	Demand	Finite	AC	ADP	Changing distribution
[21]	Divergent	Multiple		Minimize costs	Demand	Finite	AC	ADP	Dual-sourcing, Non-stationary demand
[22]	Divergent	Multiple	Lost sales	Maximize profit	Demand, Lead time	Infinite	Q-learning	RL	VMI
[23]	Divergent	Single	Hybrid	Minimize costs	Demand	Infinite	TD(λ)	NDP	
[24]	Linear	Single	Backorders	Minimize costs	Demand	Finite	Q-learning	RL	
[25]	Convergent	Single	Backorders	Minimize costs	Demand	Infinite	MARVI	ADP	
[26]	Linear	Single	Lost sales	Minimize costs, Target service level	Demand	Infinite	Action-reward	RL	Centralized and decentralized, non-stationary demand
[27]	Linear	Single		Target service level	Demand	Infinite	Q-learning	RL	VMI, Non-stationary demand, centralized and decentralized
[28]	Divergent	Single	Backorders	Target service level	Demand	Infinite	Flexible fuzzy	RL	
[29]	Divergent	Single	Lost sales	Target service level	Demand	Infinite		RL	Stationary and non-stationary demand
This thesis	General	Multiple	Backorders	Minimize costs, Target service level	Demand	Infinite			

With: [1] Carlos, Jairo, and Aldo (2008), [2] Chaharsooghi, Heydari, and Zegordi (2008), [3] Çimen and Kirkbride (2013), [4] Çimen and Kirkbride (2017), [5] Geng, Qiu, and Zhao (2010), [6] Giannoccaro and Pontrandolfo (2002), [7] Gijsbrechts, Boute, Van Mieghem, and Zhang (2019), [8] Jiang and Sheng (2009), [9] Kim, Jun, Baek, Smith, and Kim (2005), [10] Kim, Kwon, and Baek (2008), [11] Kim, Kwon, and Kwak (2010), [12] Kunnumkal and Topaloglu (2011), [13] Kwak, Choi, Kim, and Kwon (2009), [14] Kwon, Kim, Jun, and Lee (2008), [15] Li, Guo, and Zuo (2008), [16] Oroojlooyjadid, Nazari, Snyder, and Takáč (2017), [17] Rao, Ravulapati, and Das (2003), [18] Ravulapati, Rao, and Das (2004), [19] Saitoh and Utani (2013), [20] Shervais and Shannon (2001), [21] Shervais, Shannon, and Lendaris (2003), [22] Sui, Gosavi, and Lin (2010), [23] Van Roy, Bertsekas, Lee, and Tsitsiklis (1997), [24] Van Tongeren, Kaymak, Naso, and Van Asperen (2007), [25] Woerner, Laumanns, Zenklusen, and Fertis (2015), [26] Xu, Zhang, and Liu (2009), [27] Yang and Zhang (2015), [28] Zarandi, Moosavi, and Zarinbal (2013), [29] Zhang, Xu, and Zhang (2013)

We now have an overview of the research about reinforcement learning in multi-echelon inventory management. As we can see, there is already a large number of papers available. However, the papers differ substantially in their inventory systems and approaches. When we compare the different network structures of the papers, we can see that a divergent network is mostly used, followed by a linear supply chain. Only one paper uses a convergent structure and two papers describe a general network structure, which is the same network structure as CBC. Another interesting finding is that almost every paper only takes a single product into account in their method. In total, five papers define a situation with multiple products. In the case of unsatisfied demand, both lost sales and backorders are almost equally used in literature, whereas two papers consider a hybrid approach. Reaching the target service level is the goal in eight papers, most of the papers focus on the costs by either minimizing the costs or maximizing the profit. The uncertainty of demand is used in every paper, while lead time is uncertain in only three of the papers. The simulated horizon is infinite in most cases, this results in parameters for an optimal policy, which CBC is also interested in. The algorithms that are used vary, but Q-learning is the most used algorithm. Another algorithm that is used multiple times is the actor-critic (AC) algorithm, or an extended version of this algorithm, such as A3C.

As we mentioned earlier, both reinforcement learning and approximate dynamic programming are very

similar and, therefore, we look at both methods. However, it is important to distinguish it from deep reinforcement learning and neuro-dynamic programming. These methods use a neural network and substantially differ in their solution approach. While RL and ADP are both often used in inventory management, NDP and DRL are used only three times.

After reviewing these papers, we can conclude that, while reinforcement learning is used because of its promise to solve large problems, many papers still use extensive assumptions or simplifications. As we can see in the table, the problems that are solved in literature are still quite different from the current situation of CBC.

Contribution to literature

The goal of our research is to develop a reinforcement learning method for the situation of CBC. In current literature, reinforcement learning has been applied to various cases, but there is still a gap between these cases and the case of CBC. In literature, the network structures are still simple and small, while only a single product is taken into account in most cases. Next to that, reinforcement learning is praised for its versatility, but only one paper applies their method to different supply chains. Our case has a *General* network structure, and has backorders. In this research, we elaborate on existing reinforcement learning methods and expand them in such a way that we try to approach the situation of CBC. The contribution of this thesis to current literature will, therefore, be twofold: (1) We will apply reinforcement learning to a case that has not been considered before and (2) because we start by expanding current cases, we will create a simulation that can be used to apply reinforcement learning to various supply chain cases.

3.5 Reinforcement Learning in Practice

In the previous section, we have reviewed papers about reinforcement learning in inventory management. We created an overview of these papers and structured the research that has been done in this field. However, when reviewing these papers, we concluded that none of the papers has included its exact code. This means that we are not able to easily reproduce these papers and extend on their work, but have to start from scratch.

Fortunately, nowadays, blogs are a very popular way of sharing knowledge and often include practical parts, such as which software to use or pieces of code. Therefore, we also included several blog posts in this research. A popular website with high-quality blog posts is towardsdatascience.com. This website contains blogs about all kinds of machine learning and data science methods and is known for its practical solutions. At the moment, towardsdatascience.com already published over 4.500 blogs on reinforcement learning, written by Ph.D. students, researchers, developers, and data scientists. In these blog posts, the problems vary from a tic-tac-toe problem to winning a game of FIFA, but, what most blog posts have in common is that they start with an already defined environment. These environments are a part of the OpenAI Gym package.

OpenAI is a non-profit artificial intelligence research company and is founded to advance digital intelligence in a way that is most likely to benefit humanity as a whole (OpenAI, 2015). They fund interesting AI projects and encourage others to share and generalize their code. Many researchers stated that it was hard for them to reproduce their own experiments, which is why one of the first projects of OpenAI was the creation of OpenAI Gym. OpenAI Gym is created to standardize artificial intelligence projects and aims to create better benchmarks by giving a number of environments with great ease of setting up. This way OpenAI can increase reproducibility in the field of AI and provide tools with which everyone can learn about the basics of AI (Rana, 2018). In these Gyms, one can experiment with a range of different reinforcement learning algorithms and even develop their own (Hayes, 2019).

In this research, we will start with a simple supply chain and expand this model in complexity, to approach the situation of CBC and, therefore, we will model different cases. For these cases, a common and general environment will help in understanding the process, and enhance reusability and reproducibility. An OpenAI Gym would, therefore, fit this research, but, unfortunately, there is no general inventory environment available yet. At the time of writing, there is one inventory-related environment for OpenAI Gym. This gym, called 'gym-inventory', is an environment to solve a specific inventory problem with lost sales from Szepesvári (2010). Because this gym only solves a specific case about a single echelon inventory problem with lost sales, we can not use this environment for our scenario. Therefore, we will have to set up an environment ourselves. We can still use the structure of the OpenAI Gym. The gyms have a fixed structure in order to ensure the compatiblity with other projects of OpenAI, like the several deep reinforcement learning algorithms they offer. This way, it is possible to implement existing algorithms for the environment and focus on fine-tuning the algorithm instead of creating it from scratch. For this reason, we will use the structure of the OpenAI Gym and, therefore, can use their algorithms. The structure of a OpenAI Gym includes, among others, a standard definition of the action and state space and a step function that can be executed for every time step.

3.6 Conclusion

In this chapter, we have performed a literature research on reinforcement learning and multi-echelon inventory management. With this literature research, we can now answer our second set of research questions:

2. **What type of reinforcement learning is most suitable for the situation of the CardBoard Company?**
   (a) *What types of reinforcement learning are described in literature?*
   (b) *What methods for multi-echelon inventory management are currently being used in literature?*
   (c) *What types of reinforcement learning are currently used in multi-echelon inventory management?*
   (d) *How are the findings of this literature review applicable to the situation of CBC?*

In Section 3.1, we have gained more insights in the different types of reinforcement learning. We covered the elements of reinforcement learning methods and discussed several algorithms. Recently, reinforcement learning has been combined with deep learning, which was introduced in Section 3.2. This opened up new research for problems that were too large for previous methods.

Section 3.3 covers the research on multi-echelon inventory management. Multi-echelon inventory management has been researched for a longer time. Yet, this problem remains hard to solve, because of the curses of dimensionality. Therefore, most papers use extensive restrictions and assumptions, such as a deterministic demand or a limited amount of stock points. Heuristics are often used to solve the inventory problems to near-optimality, of which the base-stock policy is most popular. This policy can serve as a benchmark for our reinforcement learning method. Markov decision processes are also being used to solve inventory problems, however, none of them are applied to a general network structure.

Because of the restrictions and assumptions used in traditional methods, it is interesting to solve the multi-echelon inventory problem using reinforcement learning, which is discussed in Section 3.4. In literature, a considerable amount of research has been done on reinforcement learning in inventory management. Most of these papers focus on a linear supply chain and use Q-learning to solve this problem. General supply chains are only used in two ADP methods. Deep reinforcement learning is a recent method and therefore has not been applied broadly. At the moment, two papers have used DRL in inventory management. None of the papers included their code, which makes it hard to reproduce their results.

Therefore, we decided to look at practical examples in Section 3.5. Online, various implementations of all the reinforcement learning algorithms can be found. To ensure our code can work with these algorithms, we will use the OpenAI Gym structure, which is the most common used structure for reinforcement learning problems. By combining the algorithms from OpenAI and the knowledge and cases that we have found in literature, we will work towards a reinforcement learning method for the CardBoard Company. In the next chapter, we will introduce our first problem, for which we will implement a reinforcement learning method.

4. A Linear Supply Chain

In this chapter, we introduce a case from one of the papers in Section 4.1. The different elements of a reinforcement learning system are defined for this case, starting with the state variable and action variable in Section 4.2 and Section 4.3 respectively. Section 4.4 and 4.5 elaborate on the reward function and value function. Section 4.6 introduces the Q-learning algorithm for the reinforcement learning method. Section 4.7 covers our implementation of the simulation, where Section 4.8 expands this simulation with the Q-learning algorithm. Section 4.9 implements a deep reinforcement learning method. We end this chapter with a conclusion in Section 4.10.

4.1 Case description

As mentioned in Chapter 1, we will start by building our reinforcement learning method for a toy problem that is defined in literature. This way, we can set up a relatively simple environment and directly evaluate our solution. When this solution is fully working, we can expand the method. As a start, we chose a linear network structure, because of their limited connections and thus limited action space. Next to that, a linear supply chain makes it easier to follow the shipments, which can help in debugging the code. Although serial systems are already extensively covered in literature and can be solved to (near) optimality using mathematical models and heuristics, it is still an interesting case to test our reinforcement learning method on, because we can then easily see how the method performs. To fully reproduce the paper, ideally a paper should list its complete input and output data. This way, we can follow the method step by step, instead of only focusing on the end result of a paper. After reviewing the literature, we found one paper that fulfilled these requirements and explicitly listed the input and output data: the paper of Chaharsooghi et al. (2008). This paper uses the beer game as a case. This game is developed by Jay Wright Forrester at MIT and has a linear supply chain with four levels: retailer, distributor, manufacturer, and supplier. In the MIT beer game, the actor of each level attempts to minimize their own inventory costs, but we will focus on a system that tries to minimize the total supply chain inventory costs. The decision of ordering size depends on various factors such as supply chain inventory level, environment uncertainties, downstream ordering, and so on. In this model, environmental uncertainties include customer demand and lead times.

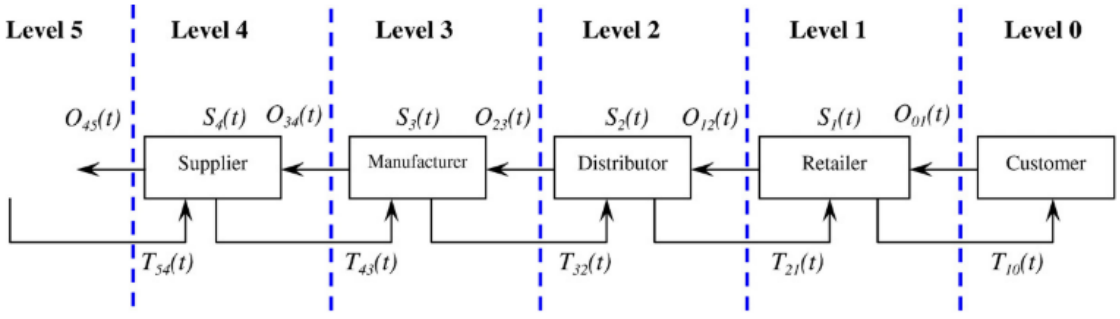


Figure 4.1: *Supply chain model of the beer game from Chaharsooghi, Heydari, and Zegordi (2008).*

According to the supply chain model shown in Figure 4.1, three groups of variables are specified to characterize the supply chain:

$S_i(t)$ represents the inventory position of level i in time step t, $(i = 1, 2, 3, 4)$

$O_{i,j}(t)$ represents the ordering size of level i to the upstream level j, $(i = 0, 1, 2, 3, 4; j = i + 1)$

$T_{i,j}(t)$ represents the distribution amount of level i to the downstream level j, $(i = 1, 2, 3, 4, 5; j = i-1)$

In this supply chain, customer demand that can not be fulfilled immediately will be backlogged and backorder costs will be incurred. These backorders will be fulfilled in the next time period if the inventory allows it. The inventory position is determined by the current inventory and the orders that will be received from upstream minus the number of backorders.

The demand of the customer is not known before the replenishment orders are placed. Next to that, the lead time is also uncertain. The lead time of any level is uncertain, except the lead time to the customer, which is always zero. For all the levels, one lead time is given per time step. Therefore, the lead time does not differ per connection between stock points, but only differs between time steps. Hence, in each time step, there are two uncertain parameters, that only become known after the stock points placed their orders.

In the beer game, the goal is to minimize the total inventory costs. There are four agents that decide on an order quantity every time period. The objective is thus to determine the quantity of $O_{i,j}$ in the way that the total inventory costs of the supply chain, consisting of inventory holding costs and penalty costs of backorders is minimized. This results in the following minimization formula, defined by Chaharsooghi et al. (2008):

$$minimize \sum_{t=1}^{n} \sum_{i=1}^{4} [\alpha h_i(t) + \beta C_i(t)] \tag{4.1}$$

Where:

$$h_i(t) = \begin{cases} S_i(t), & \text{if } S_i(t) > 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

$$C_i(t) = \begin{cases} |S_i(t)|, & \text{if } S_i(t) \leq 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

Every time step, a decision on the order quantity has to be made. This results in the following transition to the next time step:

$$S_i(t+1) = S_i(t) + O_{i,j}(t) - T_{i,j}(t) \tag{4.4}$$

In this minimization problem, $h_i(t)$ is defined as the on-hand inventory of level i at time step t, and $C_i(t)$ is defined as the backlog in level i at time step t. The latter is liable for the penalty cost. α is the inventory holding cost of each stock point, per unit per period, which is in the case of the MIT Beer Game \$1 per case per week. β is defined as the backorder cost of each stock point per unit per period, which is \$2 per case per week. n is the time horizon and is equal to 35 weeks.

At every level at each time step, four events happen. At first, the previous orders are received from the upstream stock point. Thereafter, the new order is received from the downstream level. The order is then being fulfilled from on-hand inventory if possible and put in backorder otherwise. At last, for every stock point, an order is placed upstream for stock replenishment. For the retailer, the order quantity (demand) is generated from a uniform distribution of $[0, 15]$. These steps are described later in Section 4.7.

A decision about the order size must be made for the four stock points simultaneously. This order size is then communicated to the upstream stock point. The information about the inventory position of all stock points is therefore represented in the system state in the format of a four-element vector. This state vector is used in the Q-table, together with the possible actions. This way, we have a centrally coordinated supply chain whose goal is to define a global strategy to minimize the total holding and

backorder costs. With this information, we are able to classify the beer game of Chaharsooghi et al. (2008), according to the classification of De Kok et al. (2018) and Van Santen (2019):

```
T1: 4,S,D,G|I,G|N,B|O,F,N|C|ACS,O
T2: 1||D,P,N|P,F,N|HB,N}
```

In the next sections, we will elaborate on the specific aspects of this reinforcement learning method, including the state variable, reward function, ordering policy, and algorithm.

4.2 State variable

As noted in Section 3.1, if a system state has the Markov property, the state solely depends on the state that came directly before it and the chosen action. This way, it is possible to predict the next state and expected reward given the current state and action. Even while the state signal is not absolutely Markov, we can approach it in the desired accuracy by adding an additional state variable. This way, formulating the problem as a reinforcement learning problem is possible. Since the final decision in reinforcement learning models has been made according to the system state, it is essential for the system state to provide proper information for the agents' decision-making process. The state vector defined by Chaharsooghi et al. (2008) is therefore:

$$S(t) = [S_1(t), S_2(t), S_3(t), S_4(t)] \tag{4.5}$$

Where $S(t)$ is the system state vector at time step t and each $S_i(t)$ is the inventory position of the stock point i at time step t. However, in the current definition of inventory position, the vector is infinite, which makes determining the near-optimal policy impossible because it would need infinite search power. For this reason, the inventory position is coded by mapping the state vector components to finite numbers. Chaharsooghi et al. (2008) performed a simulation run that showed a coding strategy with 9 sets is proper for their case. This coding strategy can be found in Table 4.1. This way, the infinite state size is mapped to (9^4 =) 6561 states.

Table 4.1: *Coding of the system state, extracted from Chaharsooghi, Heydari, and Zegordi (2008)*

Actual S_i	$[-\infty; -6)$	$[-6; -3)$	$[-3; 0)$	$[0; 3)$	$[3; 6)$	$[6; 10)$	$[10; 15)$	$[15; 20)$	$[20; \infty]$
Coded S_i	1	2	3	4	5	6	7	8	9

4.3 Action variable

Every time step, the order quantity has to be decided for every stock point. When ordering, the current inventory position of all the stock points is shared, but the demand and lead times are still unknown. In this case, the X+Y ordering policy is used. According to this policy, if the agent had X unit demand from the downstream stock point in the previous period, it orders X+Y units to the upstream stock point. This represents the earlier mentioned $O_{i,j}$, hence $O_{i,j}(t) = X_{i,j}(t) + Y_{i,j}(t)$. Y can be zero, negative or, positive. Hence, the agent can order equal, greater, or less than its received order. The value of Y is determined by the common learning mechanism. The action vector in period t is defined as follows:

$$Y_t = [Y_{1,t}, Y_{2,t}, Y_{3,t}, Y_{4,t}] \tag{4.6}$$

Where Y_{it} denotes the value of Y in the X+Y ordering policy for stock point i in the time step t. In the reinforcement learning method, the optimal policy is determined according to the learned value of $Q(s, a)$. Thus, for each state s the action with the greatest $Q(s, a)$ value is selected. After estimating the $Q(s, a)$ values for all combinations of states and actions, a greedy search on the Q-values can converge

the algorithm to the near-optimal policy. Hence, the policy of this reinforcement learning method can be shown as:

$$Y_S = [Y_{1,S}, Y_{2,S}, Y_{3,S}, Y_{4,S}] \tag{4.7}$$

Where Y_S is the policy in the state S. $Y_{i,S}$ is again the value of Y in the X+Y ordering policy for stock point i, but now it depends on the system state S. The optimal agent policies are determined according to the Q-values, which have to be estimated. Therefore, Chaharsooghi et al. (2008) proposed an algorithm to solve this problem. This algorithm solves the reinforcement learning model of a supply chain ordering problem and is explained in Section 4.6. The simulation performed by Chaharsooghi et al. (2008) showed that the range [0,3] is suitable for Y, which results in (4^4 =) 256 actions.

4.4 Reward function

The objective of this reinforcement learning method is to minimize the total inventory costs in the supply chain. Hence, the reward function is defined as:

$$r(t) = \sum_{i=1}^{4} [h_i(t) + 2C_i(t)] \tag{4.8}$$

Where $r(t)$ is the reward function at time step t and is a function of the inventory at a stock point and the number of backorders for a stock point at time step t. Since the MIT beer game is applied to this method, the inventory holding cost of each stock point per unit per period and backorder cost of each stock point per unit per period are set to 1 and 2, respectively.

4.5 Value function

The value function is used to find the policy that minimizes the total costs. In this value function, the next system states are embedded. Because these states are influenced by uncertain customer demand and lead time, calculating the value function is hard. Therefore, the value function will be estimated, as discussed in 3.1. In the paper of Chaharsooghi et al. (2008), the value function is written in the form denoted by Sutton and Barto (2018), being:

$$V(s) = \mathbb{E}[R_t | s_t = s] \tag{4.9}$$

Where R_t is defined as follows:

$$R_t = \sum_{k=0}^{34-t} \gamma^k \sum_{i=1}^{4} [h_i(t+k+1) + 2C_i(t+k+1)] \tag{4.10}$$

However, for clarity and consistency, we will write the value function in the form of the Bellman equation (Equation 3.2). Therefore, our value function of the beer game is denoted as follows:

Where $r(t)$ is the reward function of Equation 4.8. In this model, discounting is not considered and the gamma is therefore equal to 1. Estimating the value function will be done using Q-values for each state-action pair. The best action for each state will be specified and the optimal policy can be derived. In this model, Q-values are estimated by Q-learning, which will be explained in the next section.

4.6 Q-learning algorithm

In the model of Chaharsooghi et al. (2008), Q-values are estimated by an algorithm based on Q-Learning. The Q-values are stored in a large table, where the number of rows is defined by the number of actions and the columns by the number of states. The algorithm can be found in Algorithm 3 and will be further explained in this section.

Algorithm 3: Algorithm based on Q-Learning (Chaharsooghi, Heydari, & Zegordi, 2008).

1 set the initial learning conditions including:
2 iteration=0, t=0, Q(s,a)=0 for all s,a;
3 **while** *iteration* $\leq$ *MAX-ITERATION* **do**
4 set the initial supply chain conditions including:
5 $S_i, i = 1, 2, 3, 4$: the initial/starting inventories for the MIT Beer Game;
6 **while** $t<n$ **do**
7 with probability $Pr_{exploitation,iteration,t}$ select an action vector with maximum Q(s,a), otherwise take a random action from action space;
8 calculate $r(t+1)$;
9 **if** *the next state is s'* **then**
10 update $Q(s,a)$ using $Q(s,a) = Q(s,a) + \alpha[-r + \max_{a'} Q(s',a') - Q(s,a)]$;
11 do action a and update the current state vector;
12 increase the $Pr_{exploitation,iteration,t}$ according to the scheme;
13 $t = t + 1$;
14 $t = 0$;
15 $iteration = iteration + 1$;
16 by a greedy search on the Q(s,a), the best action in each state is distinguished;

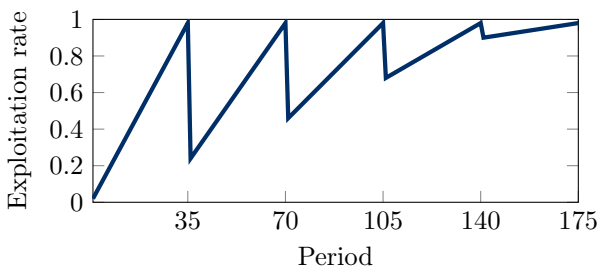


Figure 4.2: *Exploitation rate per period*

In the algorithm, we first initialize the learning conditions and create a Q-table. In the beginning, the agents do not know about the value of each action in each state, thus the initial value of all Q-values is set to zero for all state–action pairs. Every iteration, we initialize our supply chain of the beer game.

We then simulate our beer game for n periods, which is 35 weeks in this case. Every week, we select an action that has to be taken, in other words, the Y in our X+Y ordering policy. This action is either selected at random or by retrieving the highest Q-value for the current state, based on the exploitation rate. The probability of exploitation is a function of the iteration number and is increased with every iteration number linearly. Also, in each specific iteration, it is increased during period 1 to period n linearly. In the model of Chaharsooghi et al. (2008), the exploitation rate starts at 2% and is increased to 90% in the last iteration. In each specific iteration, it is increased from the start probability that is determined based on the iteration number to 98% in the nth period. Figure 4.2 shows an example of the exploitation rate with 5 iterations and 35 periods in every iteration.

When the action is taken, the simulation runs for one time step and executes the four previously mentioned events. A new state is returned, along with its reward. This reward is calculated using Equation 4.8 and contains the holding and backorder costs. In the algorithm, the value of the Q-functions is learned in every time step, hence the Q-function is estimated for every state that the agent gets to. The value is learned according to the general Q-learning formula. However, because the aim of the beer game is to minimize the inventory costs of the whole supply chain, the reward function must be minimized, which results in a negative r in the formula that is given in line 10 of Algorithm 3. This equation updates the state-action pair values in the Q-table. α is the learning rate and must be defined between 0 and 1. The

learning rate controls how much weight must be given to the reward just experienced, as opposed to the old Q-estimate. A very small α can throwback the convergence of the algorithm and a very large α (near to 1) can also intensify the effect of a biased sample and throwback the convergence of the algorithm. The simulation of Chaharsooghi et al. (2008) showed that a learning rate of 0.17 is suitable for the current case. By running the reinforcement learning method long enough, every state-action pair can be visited multiple times and the Q-values converge. After finishing the learning phase, the best action in each state is retrievable by a greedy search on each state through the Q-table. Because the beer game uses a finite horizon, the state variable is depended on the time step, as is also denoted in Equation 4.2. Therefore, $Q(s, a)$ is sometimes also denoted as $Q(s_t, a)$ in finite horizon cases, while $Q(s', a')$ is then denoted as $Q(s'_t, a')$. As this time step is already included in the state variable itself, we decided not to use the latter notation, for notational brevity.

4.7 Modeling the simulation

We now have extracted all the necessary information from the paper of Chaharsooghi et al. (2008) to replicate their reinforcement learning method. Because reinforcement learning always consists of both a simulation and optimization part, we first want to reproduce the simulation and validate this part. As mentioned earlier, Chaharsooghi et al. (2008) included both the input and output data in their paper. The policy, in terms of the Y values, that they gained using Q-learning, is therefore also stated. We can use this policy as fixed input for our model, without having to model the Q-learning algorithm. This way, we can easily compare our results with the results of the paper. We will elaborate on the events for every time step and validate the model we built. When the simulation model is validated, we can implement the q-learning algorithm, and, therefore, complete the optimization part. As mentioned before, the simulation executes four events every time step. Although these events were not further specified by Chaharsooghi et al. (2008), we were able to reconstruct what happens in every event. This was done by looking at the output that was given in the paper and by collecting information about the beer game.

At first, the previous shipments are received from the upstream stock point. The pseudo-code of this action can be seen in Algorithm 4. In this event, we look for all the stock points if there is a shipment arriving for them in the current time step. We do not have to loop over the customers, as their shipments will be delivered immediately in event 3.

When the shipment is added to the inventory, demand is generated and we can receive new orders from the downstream stock point, according to Algorithm 5. We first have to check for all stock points and suppliers if there is enough inventory to fulfill the order from the downstream stock point. Whenever this is the case, the order can be fulfilled completely. Else, we can fulfill the on-hand inventory (if any) and add the remaining amount to the backlog. Thereafter, we check if there are any backorders that can be fulfilled, using the same logic.

The fulfillment of the orders can now happen, according to event 3, given in Algorithm 6. This event is activated by event 2 and receives information about the source, destination, and quantity of the order.

At last, for every stock point, a decision has to be made about the order for its replenishment. As mentioned, this case uses an X+Y ordering policy, in which X is the incoming order, and Y is the safety factor. This Y can be either negative, positive, or zero. Based on the exploitation rate, Y will either be the best action according to the Q-value or a random action. The pseudo-code of this event is given in Algorithm 7.

Algorithm 4: Event 1. Previous shipments are received from the upstream stock point

```
for i=0 to all stock points do
    for j=0 to all stock points and supplier do
        add the shipment of current time step t from j to i to the inventory of i
        remove the shipment from in_transit table
```

Algorithm 5: Event 2. Orders are received from downstream stock point

```
generate the random demand for customer from uniform interval [0, 15]
for i=0 to all stock points and customer do
    for j=0 to supplier and stock points do
        if inventory of j ≥ order from i to j then
            fulfill the complete order                          /* See Algorithm 6 */
        else
            fulfill the on-hand inventory if possible           /* See Algorithm 6 */
            add remaining part of the order to backlog
for i=0 to all stock points and customer do
    for j=0 to supplier and stock points do
        if there are backorders for j and inventory of j > 0 then
            if inventory is larger than backorders then
                fulfill the entire inventory                    /* See Algorithm 6 */
                empty the backlog
            else
                fulfill on-hand inventory                       /* See Algorithm 6 */
                remove fulfilled amount from backlog
```

Algorithm 6: Event 3. Fulfill the order from inventory

Data: source : integer, destination : integer, quantity : integer

```
remove the quantity from source inventory
if destination is a customer then
    add quantity to customers inventory
else
    draw a random lead time according to the uniform interval [0, 4]
    ship quantity to arrive at current time step t + lead time
```

Algorithm 7: Event 4. Place an order upstream

Data: Y : list of integers

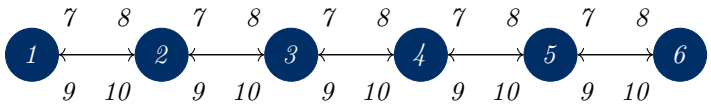
```
for i = 0 to all stock points do
    add all the incoming orders for stock point i to X_i place an order upstream with amount X_i + Y_i
```

We now have reconstructed all the events that are taken in the simulation and are able to implement these events ourselves. We modeled our simulation in Python, because of its popularity in the machine learning domain and its ease of use. The code of this simulation can be found in Appendix A.

Custom settings that have to be taken into account according to Chaharsooghi et al. (2008), is that the starting inventory of all stock points is 12 units. Hence, according to the coding of the system state as

denoted in Table 4.1, our starting state variable is $S(0) = [7, 7, 7, 7]$. The stock points also have 8 units in the pipeline, of which 4 will be delivered in time step 0 and 4 in time step 1. With this information, we can validate our simulation model. As input, we use the customer demand, lead times, and policy denoted by Chaharsooghi et al. (2008). The policy is given in the paper as output data, along with the inventory position of every stock point per time step and its corresponding costs for all 35 weeks.

To see what is going on in our simulation model and to ease the debugging, we also added a visualization. This visualization is illustrated in Figure 4.3. For every action in every time step, this visualization is generated. This way, we can easily walk through all the steps and see if our simulation matches the output from Chaharsooghi et al. (2008). In the visualization, every stock point is denoted as a separate node, while the numbers above and under the connections reflect the number of orders, the amount of backlog, the amount in transit and arriving orders. For example, the left 7 in Figure 4.3 represents the number of products that node 2 will receive from node 1 in this time period. An additional benefit of this visualization is that it will be easier to explain the code to others.



Where 1: Producer, 2: Supplier, 3: Manufacturer, 4: Distributor, 5: Retailer, 6: Customer, 7: The number of products that the downstream actor will receive this time period, 8: The total number of products that are in transit, 9: The order size that is placed upstream, 10: The amount of backorders at the upstream actor.

Figure 4.3: *Visualization of the Beer Game simulation.*

To validate the model we built, we have to compare the generated inventory position to the inventory position that is given in the paper. As we eliminated all randomness in the model, because we used the input data from the paper, the inventory positions should match exactly. However, when comparing the results, we notice two differences.

The first thing we noticed was that the inventory position stated in the output of Chaharsooghi et al. (2008), does not take the incoming orders in the pipeline into account, in contrast to their definition of the inventory position. Hence, the inventory positions are calculated solely by the number of items in stock minus the backlogged orders. The second difference happens when the backlogged orders are fulfilled whenever the inventory is larger than the number of backorders, as denoted in line 15 of Algorithm 5. When we follow the output of Chaharsooghi et al. (2008), we see that in this case, the orders are removed from the backlog, but never arrive as inventory downstream. When we adjust our model to nót fulfill the inventory and only empty the backlog, and to not take the incoming orders into account in the inventory position, the inventory positions of the created model exactly correspond to the inventory positions from the paper. We can now say that we successfully validated our implementation of the model with the model that is described in the paper.

While the first difference in the inventory positions could be a design choice, the second difference only happens in a specific case, where we can not think of a logical explanation. We, therefore, think that this is an error in the model of Chaharsooghi et al. (2008) and is not intended. While we think that our implementation of the model is correct, we want to experiment with the above two settings in order to see the implications. This results in three different experiments, on which we will run the Q-learning algorithm.

4.8 Implementing Q-learning

After successfully validating the model, we can now implement the Q-learning method. This Q-learning algorithm is stated in Algorithm 3. As documented in the algorithm, we first initialize all parameters and the Q-table. In this case, the Q-table has a size of $9^4 * 35$ (the number of states multiplied by the total time steps) by 4^4 (the number of actions). This results in a large table with 60.853.275 cells in total. According to the algorithm, we execute a predetermined number of iterations.

Almost all parameters needed for the Q-learning algorithm are mentioned in the paper, such as the learning rate and discount rate. However, the only parameter that is missing is the total number of iterations Chaharsooghi et al. (2008) has performed. For our implementation, we, therefore, have to think of an appropriate number of iterations ourselves. Because we want to run several experiments, it is important to take the running time into account. Running the reinforcement learning method for 1 million iterations takes about 2 hours. As stated before, the Q-table has more than 60 million different state-action combinations that are all possible to visit. Hence, visiting all the state-action pairs is not possible due to the lack of time. Therefore, we decided to run the experiments using 1 million iterations. We also ran the method for a longer period of 10 million iterations, but we did not see a significant improvement. Our implementation of this Q-learning based algorithm can be found in Appendix B.

Chaharsooghi et al. (2008) performed experiments using four different datasets. Although the complete output was only given for the first dataset, which we used for validating our simulation, the input data, and the results for all these four datasets are listed. Table 4.2 lists the four different datasets that are used as input for the experiments. As we can see, dataset 1 and 2 both use the same lead times as input, just like dataset 3 and 4. For the customer demand, dataset 1 and 3 share the same input.

Table 4.2: *Four test problems, extracted from Chaharsooghi, Heydari, and Zegordi (2008)*

Dataset	Variable	Data (35 weeks)
Set 1	Customer demand	[15,10,8,14,9,3,13,2,13,11,3,4,6,11,15,12,15,4,12,3,13,10,15,15,3,11,1,13,10,10,0,0,8,0,14]
	Lead-times	[2,0,2,4,4,4,0,2,4,1,1,0,0,1,1,0,1,1,2,1,1,1,4,2,2,1,4,3,4,1,4,0,3,3,4]
Set 2	Customer demand	[5,14,14,13,2,9,5,9,14,14,12,7,5,1,13,3,12,4,0,15,11,10,6,0,6,6,5,11,8,4,4,12,13,8,12]
	Lead-times	[2,0,2,4,4,4,0,2,4,1,1,0,0,1,1,0,1,1,2,1,1,1,4,2,2,1,4,3,4,1,4,0,3,3,4]
Set 3	Customer demand	[15,10,8,14,9,3,13,2,13,11,3,4,6,11,15,12,15,4,12,3,13,10,15,15,3,11,1,13,10,10,0,0,8,0,14]
	Lead-times	[4,2,2,0,2,2,1,1,3,0,0,3,3,3,4,1,1,1,3,0,4,2,3,4,1,3,3,3,0,3,4,3,3,0,3]
Set 4	Customer demand	[13,13,12,10,14,13,13,10,2,12,11,9,11,3,7,6,12,12,3,10,3,9,4,15,12,7,15,5,1,15,11,9,14,0,4]
	Lead-times	[4,2,2,0,2,2,1,1,3,0,0,3,3,3,4,1,1,1,3,0,4,2,3,4,1,3,3,3,0,3,4,3,3,0,3]

We run our reinforcement learning method for 1 million iterations. To measure the performance of the Q-learning algorithm over time, we simulate the policy that the method has learned so far, for every 1000 iterations. For these simulations, we use the datasets from Table 4.2. This way, we can see the improvement of our Q-learning implementation and can compare our results with the paper. To provide representative results, we repeat the whole procedure 10 times. The experiments are run on a machine with an Intel(R) Xeon(R) Processor CPU @ 2.20GHz and 4GB of RAM. The results of the experiments can be found in Figure 4.4.

As we can see in Figure 4.4, we run the experiments using three different experimental settings. In these experimental settings, we varied with the fulfillment of backorders according to line 15 in Algorithm 5 and the definition of inventory position, where we either do or do not take the incoming shipments into account. Experiment 1 represents the settings with which we validated our simulation model with the one from Chaharsooghi et al. (2008). Hence, it does nót fulfill the shipments according to line 15 in Algorithm

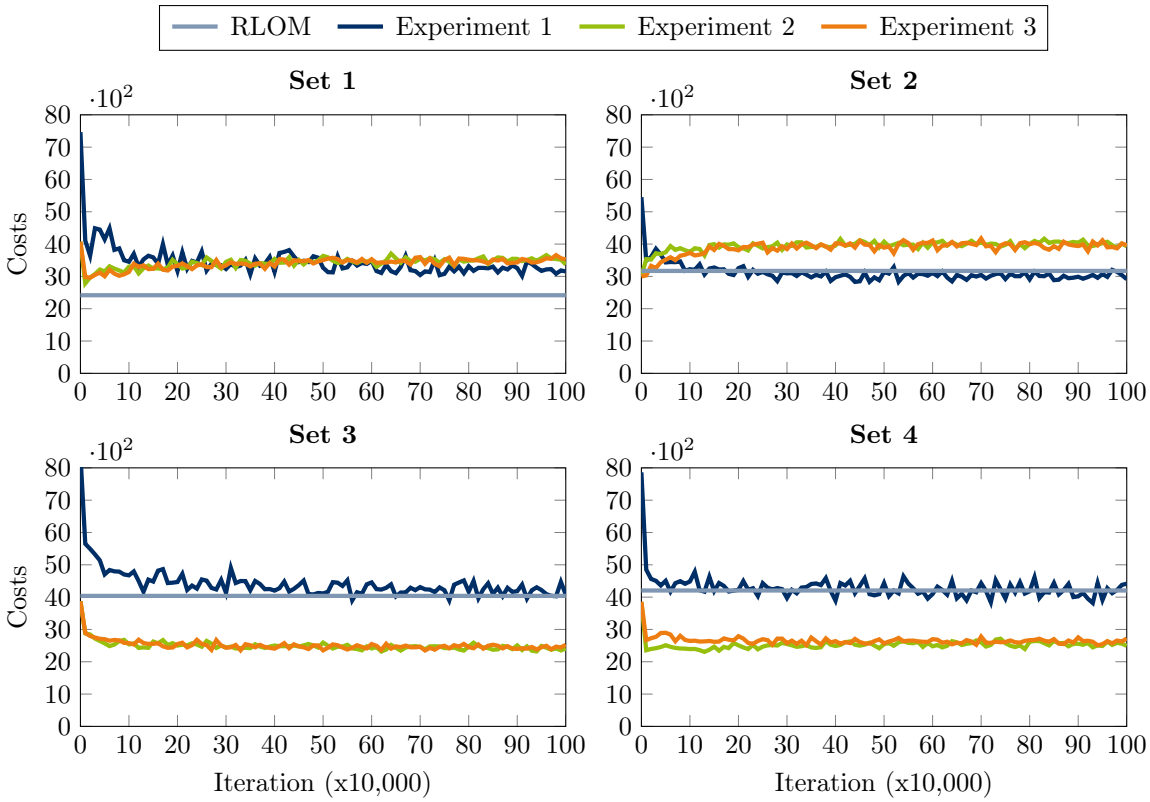


Figure 4.4: *Costs of the Q-learning algorithm per dataset. Lower is better.*

5 and does nót take the incoming shipment into account in the inventory position. Experiment 2 does fulfill the shipments but does nót take the incoming shipments into account. Experiment 3 does fulfill the shipments ánd take the incoming shipments into account in the inventory position. Although we know for sure, through the validation of our simulation model, that the settings of Experiment 2 and 3 are not used by Chaharsooghi et al. (2008), and can therefore not be used to compare these results to the RLOM, we still use these settings, because we want to know if the Q-learning method is able to learn better using these settings. The result from the paper, where the method is called the Reinforcement Learning Ordering Mechanism (RLOM), is also visualized in this figure to easily compare the results. Further on, we will compare the results in more detail. Because we only have the end results from the paper, we visualize them using a horizontal line. Please note that the figure visualizes the costs and, therefore, uses positive numbers.

We can immediately see that Experiment 1 approaches the result of the RLOM in three of the four datasets. However, for dataset 1, the RLOM yields a significantly better result than our method. We can see an improvement over the first few iterations for our method, but after these few iterations, no significant improvement is gained.

We also performed Experiment 2 and 3 to see if these experiments were able to learn better. Although these experiments yield better results on datasets 3 and 4, they perform worse on dataset 1 and 2. We can therefore conclude that the method is not able to learn better with these settings and the improvement on datasets 3 and 4 are most probably because of the fulfilled shipments. As mentioned earlier, we will not compare these experiments with the RLOM, because it uses different settings and, therefore, can not be compared.

Although approaching the results in three of the four datasets is a good results, we are still curious to

see why we are not able to match the result for dataset 1. To gain insights into the origin of the rewards of the different experiments, we visualize the costs per time step. For each dataset and experiment, we use the final policy to visualize the costs that are made over time. The costs over time per dataset can be found in Figure 4.5. In this figure, holding costs are denoted as positive integers, while backorder costs are denoted as negative integers, to simplify the visualization. For dataset 1, we have included the holding and backorder costs of the paper. This data was not available for the other datasets, and is therefore not included in the other graphs. We can see that the holding and backorder costs of the paper match the costs of Experiment 1 most of the time, which indicates that our initialization and method are almost certainly implemented the same way as in the paper of Chaharsooghi et al. (2008). We can also see that Experiments 2 and 3 often have more inventory than Experiment 1. While this results in larger holding costs for dataset 1 and 2, it decreases their backorder costs in dataset 3 and 4. The reason for the higher inventory of Experiments 2 and 3 can be found in the fact that, as opposed to Experiment 1, the shipments are fulfilled according to line 15 of Algorithm 5, as mentioned earlier. This results in 'lost' shipments for Experiment 1, because the backorder is reduced, but the items are never added to the inventory. In Experiments 2 and 3, these shipments áre added to the inventory, which results in the higher inventory overall.

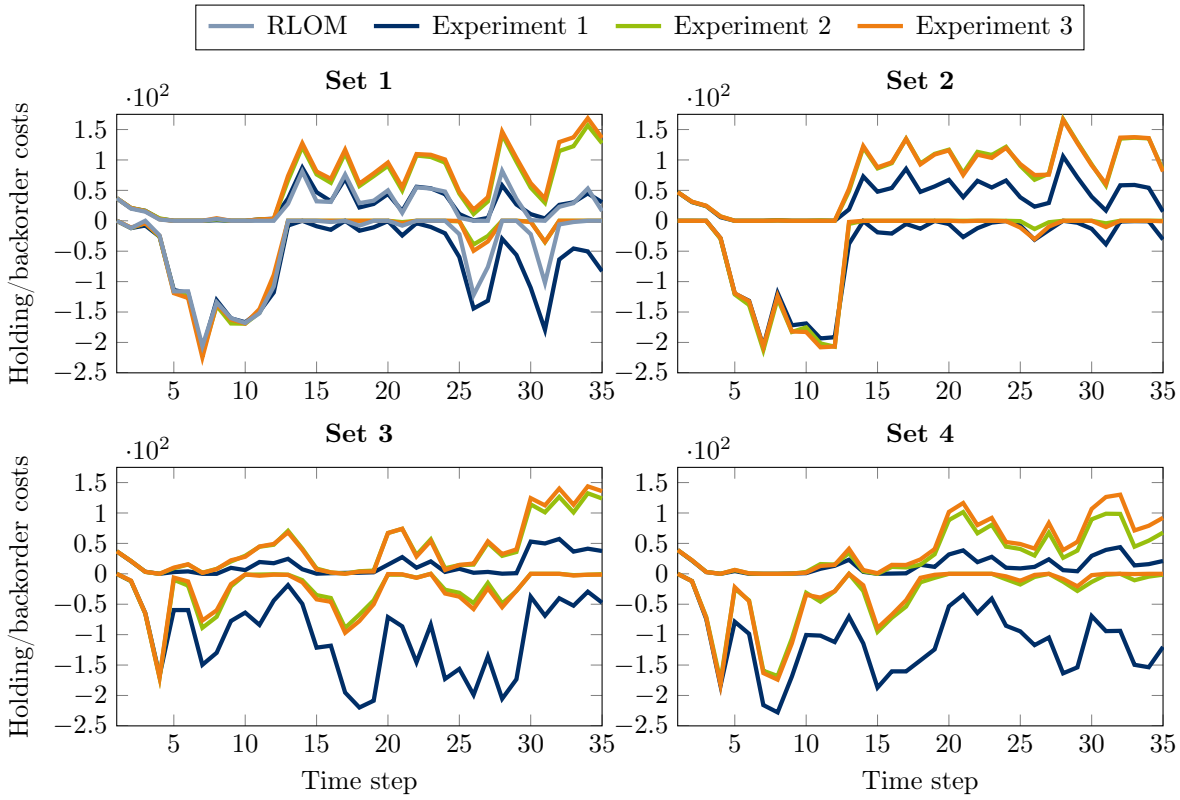


Figure 4.5: *Costs over time per dataset. Closer to zero is better.*

In the paper of Chaharsooghi et al. (2008), two different methods are used as a benchmark. The first method is the 1-1 ordering policy, where each stock point orders to the upstream stock point whatever is ordered from downstream. Hence, it is the same as the X+Y ordering policy, with the Y being zero every time. The second method is an ordering policy based on a Genetic Algorithm (GA) and is introduced by Kimbrough, Wu, and Zhong (2002). These two methods are used to compare to the results of the RLOM, introduced by Chaharsooghi et al. (2008). In Figure 4.6, we visualize the costs that are given in Chaharsooghi et al. (2008) and compare them with the end results of our method. We can see that the RLOM also greatly differs in results in the various datasets. When we compare the RLOM with

our Experiment 1, we can see that we are able to approach the results of the RLOM, except for dataset 1.

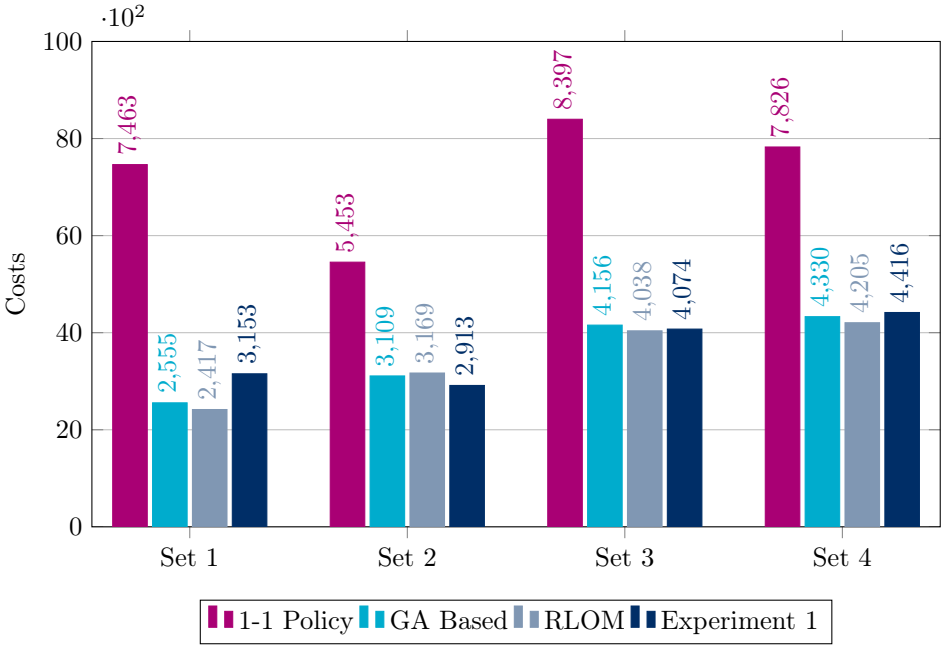


Figure 4.6: *Comparison of the results from the paper and our results. Lower is better.*

We can say that our implementation of the reinforcement learning method is able to approach the results of the RLOM of Chaharsooghi et al. (2008). However, this is not yet a confirmation that the Q-learning algorithm is indeed able to learn the correct values. To check this, we take a look at the Q-values. A Q-value represents the expected reward of the current state and its following states, until the end of the horizon, corresponding to the value function. Hence, when the algorithm is successfully able to learn these rewards, the highest Q-value of the first state in the horizon should approximately be the same as the total rewards in the horizon. The course of this highest Q-value of the first state per iteration is visualized in Figure 4.7.

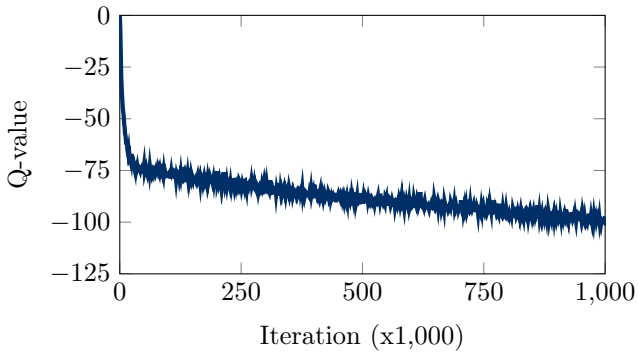


Figure 4.7: *Highest Q-value of the first state per iteration.*

The Q-value is a negative number, because the rewards are costs that we want to minimize, which is also stated in Algorithm 3. As can be seen, the Q-value differs from the total rewards given in Figure 4.6 and does not come close to it. The Q-values are initialized with zeros and decrease fast in the first iterations, but later on follow a linear pattern. We can explain this linear pattern by the fact that the Q-values are initialized with zeros and the current number of iterations is not enough to visit all state-action pairs. This way, the maximum value of the future state-action pair, denoted by $\max_{a'} Q(s', a')$, is often zero. To prove this, and to ensure that our our implementation of the Q-learning algorithm is correct, we experimented with a smaller state-action space and noticed that the highest Q-value of the first state did indeed correspond to the final reward. The results of this experiment can be found in Appendix C. Because the Q-values are initialized with

zero, the Q-learning algorithm assumes that a state-action pair it has not visited yet, is always better than the state-action pairs it has already visited. This makes the algorithm, especially in combination with the current exploitation rate, exploration-oriented. In a case with only a few state-action pairs, this is not really a problem, because all the states can be visited multiple times. In our case, it could be that the results shown in Figure 4.6 are based on exploration, instead of exploitation of the known values. Therefore, we propose three different adjustments to the current Q-learning algorithm: initialized values, a harmonic stepsize and an epsilon-greedy policy.

Initialized values

In our case, we want to minimize our costs, while Q-learning algorithms try to maximize the Q-values by default. Therefore, we take the costs into account as a negative number, so we can indeed maximize the Q-values. The Q-values are often initialized with a value that can easily be improved by the algorithm. For example, if we had positive rewards that we wanted to maximize, it would make sense to initialize the values with zero, because the Q-learning algorithm could easily improve that. As mentioned earlier, our Q-values are also initialized with zero, which is, in our case, a high value that is impossible to reach, because it would mean we have a time period without any (future) costs. This way, unvisited state-action pairs always seem like the best choice for the algorithm to visit, which makes it exploration-oriented. In our case, it would therefore make sense to initialize the Q-values with a negative value that can easily be improved. We chose to set this value on -8000. This value can be gained using a simple 1-1 policy, as can be seen in Figure 4.6, and can therefore easily be improved. To speed up the learning process, we also implement a linear increase in the initialized Q-values over the time horizon. This way, the Q-values of the state-action pairs of the first time step are initialized to -8000, with a linear increase to the state-action pairs of the last time step. These are initialized with -350, which is in line with the 1-1 policy.

Harmonic stepsize

The stepsize is used in the calculation of the Q-value and is the weight that is given to a new observation relative to to the current Q-value. This stepsize is denoted by α in Algorithm 3 and has a value of 0.17 in the paper of Chaharsooghi et al. (2008). Although there is no optimal stepsize rule, Powell (2011) suggests a range of simple stepsize formulas that yield good results for many practical problems. We apply one of these rules, a generalized harmonic stepsize (Mes & Rivera, 2017; Powell, 2011):

$$\alpha = \max\{\frac{a}{a+n-1}, \alpha^0\} \tag{4.11}$$

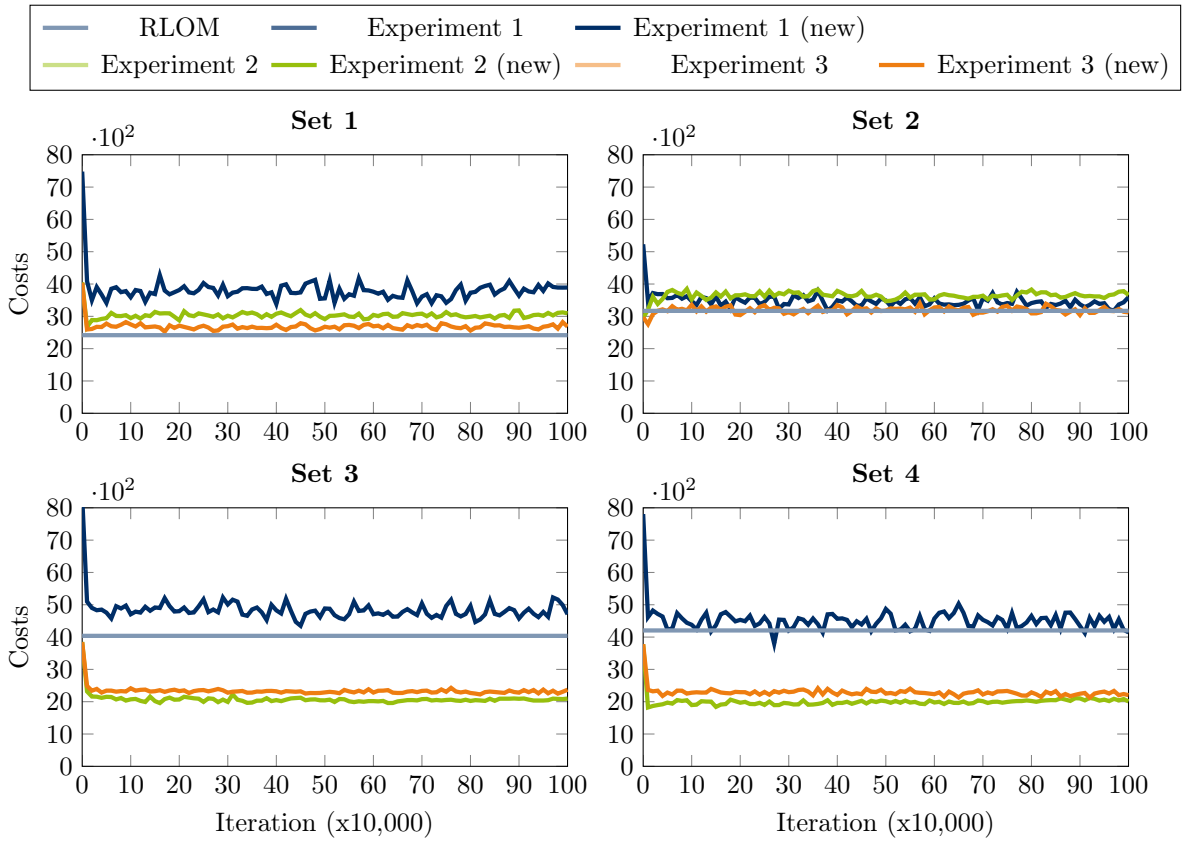
Where n is the current iteration, a can be any positive real number and $\alpha^0 = 0.05$. The advantage of the harmonic stepsize is that a larger weight is given to the rewards in the first iterations, and therefore a smaller weight to the existing q-values. This makes sense, because these q-values are the initialized values, instead of actual observed rewards. Increasing a slows the rate at which the stepsize drops to zero. Choosing the best value for a therefore requires understanding the rate of convergence of the application. It is advised to choose the a so that the stepsize is less than 0.05 as the algorithm approaches convergence (Powell, 2011). Following the formula of Mes and Rivera (2017), we define a minimum α, being 0.05. Because we run our method for 1 million iterations, we set a to be 5000.

Epsilon-greedy policy

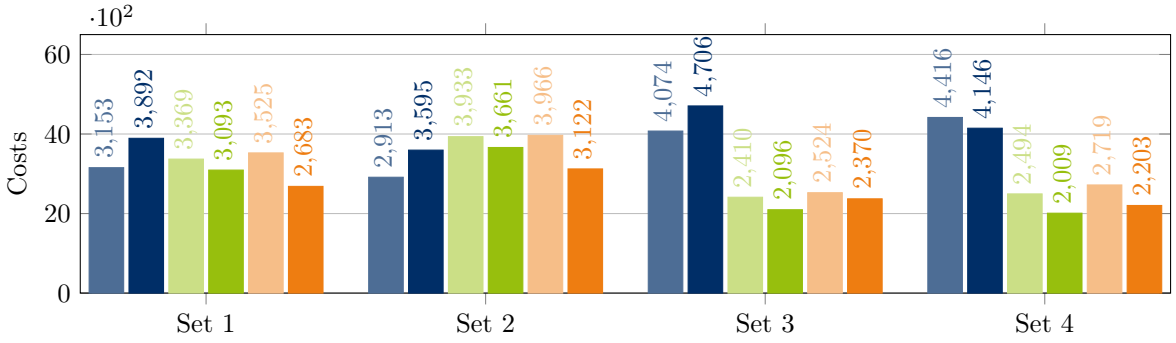
With the new initialized values, our Q-learning algorithm is exploitation-oriented. In combination with the exploitation rate as defined in the paper of Chaharsooghi et al. (2008), our method is too focused on exploitation and does rarely explore. Because we want to encourage our method to explore unvisited states, we also implement a new exploitation-exploration logic. The balance between exploration and

exploitation is not trivial, but an often used method is the epsilon-greedy policy. In this policy, we set a fixed epsilon, where $0 < \varepsilon < 1$. Every time step, we choose a randomly selected action with probability epsilon, or go with he greedy choice otherwise (Szepesvári, 2010). With the epsilon-greedy policy, we maintain a certain degree of forced exploration, while the exploitation steps focus attention on the states that appear to be the most valuable (Powell, 2011). We set the epsilon to have a value of 0.05.

With these three adjustments to the algorithm, we can run the experiments again and evaluate the results. We also included Experiments 2 and 3 again, because we were curious to see if the new method was able to improve for these settings. Figure 4.8a shows the results of the Q-learning algorithm per dataset and Figure 4.8b shows the comparison of the new Q-learning algorithm with the old algorithm.



(a) Results of the revised Q-learning algorithm per dataset. Lower is better.



(b) Costs using the Q-learning algorithm and the revised Q-learning algorithm. Lower is better.

Figure 4.8: *Results of the revised Q-learning algorithm.*

Looking at the results of the different experiments in Figure 4.8, we can see that the results of Experiment 1 did not improve with the new Q-learning algorithm, and is, despite the alterations to the Q-learning

algorithm, performing slightly worse. Experiments 2 and 3 were able to improve their results, which confirms that the new method is indeed an improvement, but the method is just not able to learn correctly with the settings of Experiment 1. Because we expected the algorithm to perform better with the alterations, we dive further into the Q-values.

Figure 4.9 shows the highest Q-value of the first state per iteration. As can be seen, the Q-values of the new Q-learning algorithm do approach the average results. The highest Q-value of the first state in the last iteration, for Experiment 1, is -4250, while the rewards of this experiment vary around -4000. However, when we take a closer look at all the different Q-values of the first state, which can be found in Appendix D, we notice that the values do not differ much. This means that, according to the algorithm, there is no obvious right or wrong action to choose, which results in an almost random policy.

We also noticed that these Q-values, and therefore also the best action, still vary over time, resulting in an almost random choice of actions. To investigate whether this is indeed the case, we performed a simulation where we selected a random action for every time period. We tested the results on the different datasets and used 100 replications. These results can be found in Figure 4.10. As we can see, the random actions yield a result that is almost equal to the results of the paper, except for dataset 1. We can, therefore, conclude that, although the paper promises to learn the optimal state-action pairs, it does not even manage to beat a method using random actions. We suspect that the results for dataset 1 are used from a specific run that only manages to yield good results by coincidence.

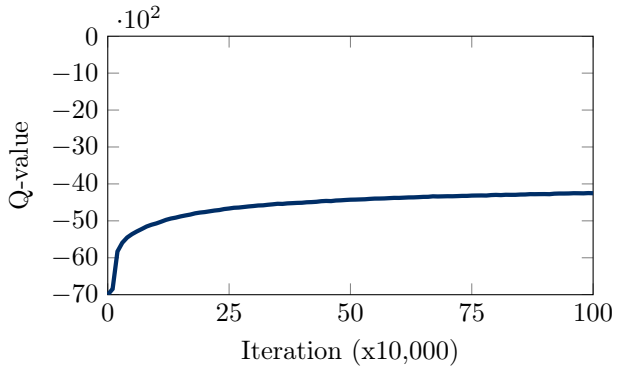


Figure 4.9: *Highest Q-value of the first state per iteration.*

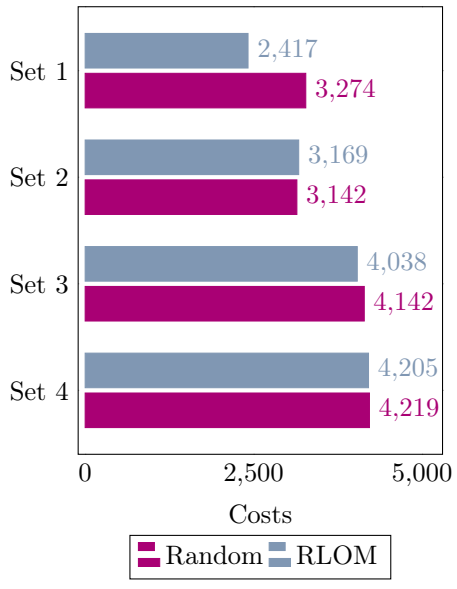


Figure 4.10: *Costs using the RLOM and random actions. Lower is better.*

However, this does also mean that our revised Q-learning algorithm did not succeed in learning the best actions for this inventory problem. We can think of three main reasons why the algorithm did not succeed, being a small action space, the coded state space and the large randomness of the environment. In the environment, we have a random demand and random lead time, which both have a large impact on the size and arrival time of the order. When ordering upstream, the order can arrive 0 to 4 weeks later, which is a large difference, especially considering the horizon of 35 weeks. The state space does not take the incoming orders into account, and needs to code the inventory and therefore generalizes certain inventory positions. This results in leaving out important information in the state vector. Concerning the action space, we defined the order policy to be $O = X + Y$. Although we can vary Y from 0 to 3 for every node, this only gives us a limited impact in the final order size, as the X is defined as the downstream demand of the previous period, which can be 0 to 24.

It would be interesting to look at a bigger action space, where we define the complete order size (O). To do this, we have to increase the size of the Q-learning table. However, we noticed that the Q-learning table used in this problem is already immense and almost reaches the memory limit of our computer. Expanding the state- or action size is not possible, because the Q-table can not be instantiated in that case. Therefore, we can conclude that the current Q-learning algorithm, because the usage of the Q-table, can only be used in cases where the inventory positions are coded and a small action space is used.

Conclusion

In this section, we implemented Q-learning. We started with implementing the algorithm of Chaharsooghi et al. (2008). We tested this algorithm on four predefined datasets and defined three different experiment settings, to determine the setting that corresponded to the simulation of Chaharsooghi et al. (2008). The first experiment setting was based on the output data of Chaharsooghi et al. (2008) and did not fulfill the shipments according to line 15 in Algorithm 5. Experiment 2 did fulfill these shipments, just like Experiment 3. Experiment 3 did also take the incoming shipments into account in the state vector. Based on the results, we were able to confirm that Experiment 1 was indeed used by Chaharsooghi et al. (2008). However, because this experiment results in 'lost shipments', we will not use this setting in the next cases, but will use the setting of Experiment 2 instead.

With the algorithm of Chaharsooghi et al. (2008), we did not get the results we expected, as there was no improvement for most of the datasets. Because we noticed there was room for improvement in the algorithm, we decided to implement these improvements, being: initialized values, a harmonic stepsize and the epsilon-greedy policy. Despite these improvements, the results did not improve on all datasets. Therefore, we decided to run the simulation while using random actions and saw that these results corresponded to the results of Chaharsooghi et al. (2008) for most of the datasets. This way, we concluded that the paper of Chaharsooghi et al. (2008) was not able to learn a good policy, but had such a small action space that even random actions yielded a result that was better than their benchmarks. We also concluded that the Q-learning algorithm with a Q-table is not scalable and, therefore, can not be used in future cases, as these cases are more complex. Therefore, we need to look for another method that can be used to solve this problem. Hence, in the next section, we will investigate if we can apply Deep Reinforcement Learning to this problem, which promises to be more scalable.

4.9 Implementing Deep Reinforcement Learning

In Chapter 3, we have elaborated on several reinforcement learning methods. We learned that for cases where we cannot explore all state-actions pairs, value function approximation can be used. There are several ways to use value function approximation in reinforcement learning, but the most popular one is the use of deep learning. We have explained several deep reinforcement learning methods and found that Proximal Policy Optimization was one of the most recent and most successful methods. Because of its success in recent applications, we will implement this method for the beer game.

We have set up our simulation of the beer game using the OpenAI Gym structure. As mentioned in Section 3.5, this structure makes it possible to use the algorithms of OpenAI and other packages that are also based on the OpenAI structure. For our implementation of the PPO algorithm, we make use of the Python Deep Learning Libraries 'Spinning Up' (Achiam, 2018), 'Stable Baselines' (Raffin et al., 2019), and 'PyTorch' (Paszke et al., 2019). PyTorch is used to build the Neural Network. Spinning Up and Stable Baselines are libraries that are used to easily implement deep reinforcement learning algorithms. Because we want to be able to make changes to the PPO algorithm, we will not directly use these libraries, but their code will serve as a foundation for our PPO algorithm. When implementing our adaption of

the PPO algorithm, we use the values of the hyperparameters from the work of Schulman et al. (2017). When implementing a neural network, there are a few design choices that have to be made. In the next subsections, we will elaborate on the choices we made.

State space

Because we are no longer limited by the maximum size of the Q-table, we can redefine our state-vector. The definition of a state is crucial to the learning of the agent. Because we want the agent to learn the expected values of a state the best way possible, it would be logical to pass all the information that is available to the network. This way, the state variables capture enough data about the current state such that the problem has the Markov property, meaning that the prediction of the future rewards and states can be made using only the current state and action choices. Hence, all the information that was previously stored inside the environment can now be exposed to the input layer of the network. For every variable in our state vector, a separate input node is created in our neural network. Our new state vector is defined as follows:

$$S = [ti, tb, h_i, C_i, d_i, tr_{i,t}, o_{i,t}] \tag{4.12}$$

As can be seen, a lot of information is added to the state vector. ti defines the total inventory in the system, tb defines the total number of backorders in the system. h_i and C_i define the inventory and backorders per stock point respectively. d_i is the demand for the previous period per stock point. $tr_{i,t}$ is the total number of items that currently are in transit per stock point per time step, while $o_{i,t}$ is the number of items that are arriving in time step t for stock point i. We have decided to include the time step for these last two variables for a horizon of 4, because of the maximum lead time, which is also 4. Hence, for $o_{i,1}$ it takes 1 time step for these items to arrive, for $o_{i,4}$ it takes 4 time steps for the items to arrive.

The values of these state variables are normalized to a range of $[-1, 1]$, to keep the values close together. A neural network is initialized with small values and updated with small gradient steps. If the input values are not normalized, it takes a long time to step the weights of the neural network to correct values. We want the interval of the state variables before normalizing to have a wide range, to ensure we do not limit our method in its observations. The lower bound is 0 for every variable because we do not have negative orders, backorders, or shipments. The upper bound of the values before normalizing can be found in Table 4.3.

Table 4.3: *Upper bound of the state variables for the beer game.*

Variable	Upper bound
ti, tb	4000
h_i, C_i	1000
tr_{it}	150
d_i, o_{it}	30

Action space

Reinforcement learning problems may either have a discrete or continuous action space. For every output node, the neural network of the actor outputs either the chance that this action will be executed (discrete actions) or the action-value (continuous actions). With a discrete action space, the agent decides which

distinct action to perform from a finite action set. For every distinct action in this action set, the neural network has a separate output node. In the beer game case of Chaharsooghi et al. (2008), this would match the number of rows in our Q-table, being $4^4 = 256$ nodes. With a continuous action space, the number of output nodes of the neural network is equal to the length of the action set. In the beer game case, this would be 4, corresponding to the number of stock points for which we want to take an action. The output of the neural network is a mean and standard deviation for every node. These values correspond to the amount that has to be ordered for every stock point. The PPO algorithm implementation of Spinning Up works with both discrete and continuous action spaces. As mentioned in the previous section, we want to use deep reinforcement learning to estimate the complete order size, instead of using the $X + Y$ order policy. This way, the number of distinct actions can become very large. For example, when we want the order size to be in the interval of $[0, 30]$, we have $31^4 = 923.521$ different actions. Therefore, a continuous action space is recommended. This action space results in a more scalable neural network, as the number of output nodes only grows with the number of stock points, not the number of actions.

The default output of an untrained network is 0, with standard deviation 1. It could, therefore, take a long time to reach the desired mean output. For example, if the optimal order size is 20, the neural network has to adjust its output to 20, but only does this in very small steps. Hence, it is recommended to scale the action space to lie in an interval of $[-1, 1]$ (Raffin et al., 2019). This should speed up the learning of the network. For the beer game, we decided to use an action space with the interval of $[0, 30]$. As the average demand is 10, this upper bound should not limit the method. We ran some tests and concluded that it is indeed better to rescale the action to the interval of $[-1, 1]$. Hence, when the neural network has 0.4 as output, the environment rescales the action back to its original interval and therefore will order 21 units. We also experimented to scale the actions to an interval of $[0, 5]$ and, although this gave slightly better results for most of the runs, there were also runs where the method was not able to learn at all. Overall, the interval of $[-1, 1]$ yields the best results. Because the action space output of the neural network is unbounded, we clip the outputs that lie outside the given interval. This way, we encourage the network to stay within the limits of the interval. We have also experimented with giving the network a penalty whenever the outputs were outside the given interval, but this yielded worse results.

Horizon

Next to that, an important aspect is the horizon of our deep reinforcement learning implementation. The beer game case from Chaharsooghi et al. (2008) uses a finite horizon. The method is executed for 35 weeks and for every week, an action is determined. This way, an action that is performed near the end of the horizon, does not always affect the situation, because the result of this action can lie after the horizon. Often, a discount factor is used, such that immediate actions weigh heavier than actions in the distant future. In the beer game, no discount factor is used, hence, future costs are just as heavily weighted as present costs. In the Q-learning method, for every time step in the horizon, a separate table was defined. In deep reinforcement learning, the current time step would be represented by one input node, as it is one variable in the state vector. In PPO, we have two parameters that are important when setting the horizon, the buffer size and batch size. The buffer size is the length of an episode and is often also denoted as the horizon. The batch size is the size used to split the buffer to update the weights of the model. In this finite horizon, we let the buffer length and the batch size have the same length as the horizon defined by Chaharsooghi et al. (2008). However, we have conducted experiments using a finite horizon of 35 periods, but the method was not able to learn properly. Therefore, we have decided to run the method on an infinite horizon. We have removed the time variable for our state vector and include a

discount factor of 0.99. We still evaluate the performance on the finite horizon of 35 periods. The length of our buffer is set to 256, the batch size is set to 64.

Rewards

Just like Q-learning, PPO tries to maximize the rewards. Because we want to minimize our costs, we once again multiply the costs by -1 to define the rewards. As mentioned in Section 3.2.2, the PPO algorithm uses an Actor-Critic architecture. While the Actor outputs the values of the actions, the Critic estimates the value function. To estimate this value function, the neural network performs a regression on observed state-action pairs and future values. The network weights are then optimized to achieve minimal loss between the network outputs and training data outputs. However, these future values can be very large, especially in the beginning, when the network is not yet trained. If the scale of our network outputs is significantly different of weight and bias from that of our input features, the neural network will be forced to learn unbalanced distributions of weight and bias values, which can result in a network that is not able to learn correctly. To combat this, it is recommended to scale the output values (Muccino, 2019). In our experiments, we noticed that the scaling of these rewards is not trivial and no best practices are defined yet. In our case, dividing the results by 1000 yielded good results and we saw that the expected value of the Critic corresponded to the average reward that was gained in the experiments.

Neural Network

In Section 3.2.1, we explained the concept of neural networks. To use a neural network in our experiments, we have to define the number of hidden layers, number of neurons in these hidden layers and the activation function. The number of neurons of the input layer is already given by the state size, while the number of neurons of the output layer is equal to the number of stock points. In literature, several rule-of thumbs can be found to determine the size of the hidden layers. However, most of these are not trivial or only based on specific algorithms or experiments. Therefore, we have decided to stick to the default neural network size of the PPO algorithm as defined by Schulman et al. (2017), which has two hidden layers, each with 64 nodes. We have also experimented by expanding the neural net to four hidden layers or to 128 nodes per layer, but this gave worse results, mostly because the outputs of the neural network where varying more over time.

Experiments and results

Now that we have defined all the relevant settings for the PPO algorithm, we can run our experiments. The algorithm is set to train 10.000 iterations. Every 100 iterations, we perform the simulation, with the horizon of 35 time steps, to measure the current performance of the algorithm, using the learned policy. Because we do not use the method of the paper of Chaharsooghi et al. (2008) anymore, we decided not to evaluate the performance on the predefined datasets of the paper. Instead, we evaluate using realizations of the same distributions as the ones we use in the learning phase. The simulations are repeated 100 times, of which the average is used. As mentioned earlier, we will no longer use the order policy $X + Y$, but will decide on the full order size O. We have performed both Experiments 1 and 2, whose definitions remain unchanged. Hence, Experiment 1 is, once again, the method of the paper, with the lost shipments. Experiment 2 is our adaption to the simulation and does fulfill these shipments. Experiment 3 was created to include the incoming orders in the state vector, but because we have completely redefined our state vector for the deep reinforcement learning method, this experiment has become obsolete. The PPO algorithm takes about 1 hour to complete a run. In total, 10 runs are performed. The results of the experiments can be found in Figure 4.11.

As a benchmark, we have also included the average results of the RLOM. Although we now know that these results are only gained using random actions, they can still be used as a reference for what the costs should be. In fact, because of this fact, we can now also calculate a benchmark for Experiment 2. This is done by performing a simulation using the $X + Y$ order policy and random actions for 500 times, with the settings corresponding to Experiment 2. The average result is used as benchmark.

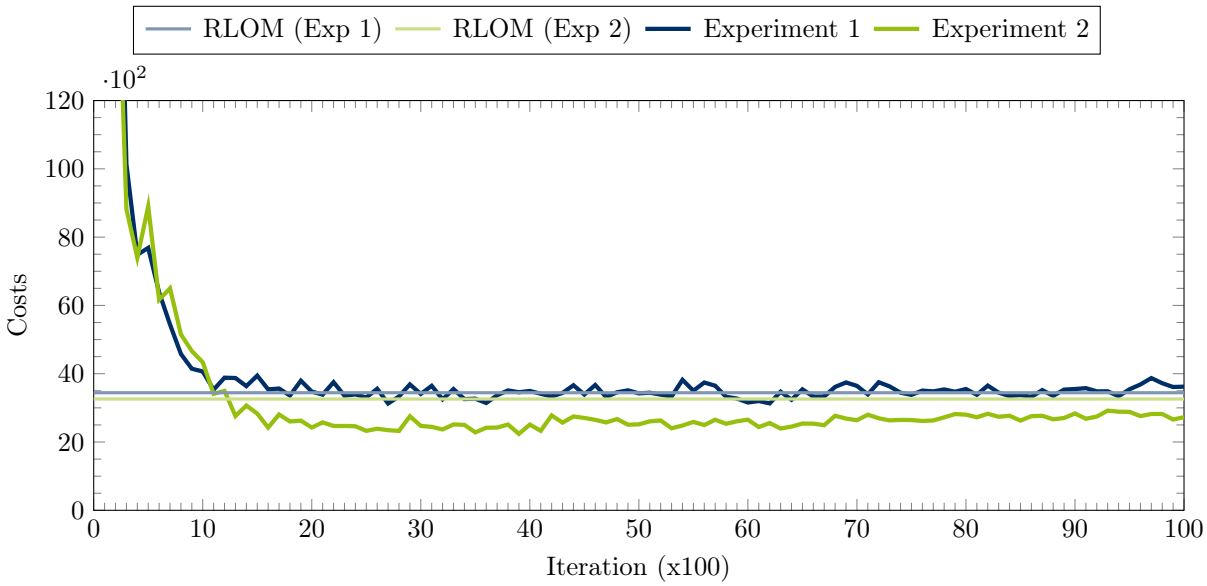


Figure 4.11: *Costs of the PPO algorithm on the beer game. Lower is better.*

Another benchmark that is often used in linear supply chains is the heuristic of Shang and Song (2003). This heuristic can be used to determine the (near-)optimal base-stock levels for every stock point in the linear supply chain. However, because the heuristic assumes a constant lead time, we were not able to use this benchmark.

Due to the large action space, the first iterations yield a poor result. However, we can see that the method improves quickly and it only needs a few iterations to get close to the results of the RLOM. Because we now decide on the full order size, it is quite impressive that the DRL method is able to approach and improve the RLOM in a few iterations. Experiment 1 is not able to significantly improve the results of the RLOM and the results vary around the results of the RLOM, which could mean that a lower reward is hard to gain in this setting. Experiment 2, however, is able to beat the X+Y ordering policy with random actions and shows less variation over the iterations, which indicates that the method is able to learn better using this setting.

Conclusion

In this section, we have successfully implemented deep reinforcement learning using the Proximal Policy Optimization algorithm. To get this algorithm working, we needed to define several parameters. We did some experiments with these parameters to see what values were desired, which led to the following findings:

- Values of the environment that concern the neural network, such as the actions and states, should be normalized. We chose an interval of [-1, 1].
- The state space should include all information that is known in the environment. By exposing this data to the neural network, we make sure that the network has the best opportunity to learn.

- If the neural network has to decide on a lot of actions, it is desirable to use a continuous action space. This action space results in a more scalable neural network.
- As defined in the PPO algorithm, the neural network is trained in batches. We yielded the best results using the default parameters of the PPO algorithm, using a buffer size of 256 and a batch size of 64.
- We now train our data using an infinite horizon. Hence, the time variable is removed from the state vector and we apply a discount rate of 0.99.
- It is recommended to also scale the rewards of the environment. This is, however, not trivial and no best practices are defined yet. To make our rewards smaller, we divided the rewards by 1000 before passing it to the neural network. This yielded better results in our experiments.
- Varying the number of neurons per layer of the neural network did not significantly impact our results. However, a two-layer network yielded better and more steady results than a four-layer network.

We can conclude that the PPO algorithm succeeded in yielding good rewards while using a large action- and state space. Because of the large action space, random actions would not yield better results than the benchmark. We, therefore, know for sure that the method is able to learn the correct actions. With these results, deep reinforcement learning is a promising method for other cases. Instead of trying to improve the results even further, by tuning the hyper-parameters, such as the buffer length and state space, we move on to the next case, which will be explained in the next chapter.

4.10 Conclusion

In this chapter, we have introduced a linear supply chain from the paper of Chaharsooghi et al. (2008). We have followed this paper to build a reinforcement learning method to minimize the total holding and backorder costs in the supply chain, in order to answer our sub-question 3a:

3. **How can we build a reinforcement learning method to optimize the inventory management at CBC?**
   (a) *How can we build a reinforcement learning method for a clearly defined problem from literature?*

In Sections 4.1 to 4.6, we have introduced the beer game. We covered the relevant variables that are needed for the Q-learning algorithm, the algorithm that was used by Chaharsooghi et al. (2008). With this information, we were able to reconstruct this paper ourselves. In Section 4.7, we successfully validated our simulation model with the model of Chaharsooghi et al. (2008), using their input and output data.

Thereafter, in Section 4.8, we implemented the Q-learning algorithm from the paper and were able to approach the results of this paper. However, when we took a closer look at the Q-values, we concluded that the Q-learning algorithm was not learning. Therefore, we suggested some improvements to this algorithm. Yet, due to the limited action space and high stochasticity, the new Q-learning algorithm did not improve the results. We discovered that the results of the paper were mostly gained using random actions. This was possible because Chaharsooghi et al. (2008) used a X+Y order policy, where the Q-learning algorithm had to decide on the Y in the interval of $[0, 3]$. The X value of this policy has an interval of $[0, 24]$. This way, the impact of the Y values, and, therefore, the impact of the Q-learning algorithm, was limited. Next to that, the Q-learning method already reached its limits with respect to the state and action sizes. Therefore, we decided to implement a new method that is able to use a larger action- and state space: deep reinforcement learning.

Deep reinforcement learning uses a neural network to estimate the value function and is therefore a more flexible and scalable method. We implemented deep reinforcement learning using the PPO algorithm in

Section 4.9. This algorithm is one of the most recent algorithms and praised for its wide applicability and good results. We have implemented the algorithm using an adaption of the packages 'Spinning Up' and 'Stable Baselines'. We were able to successfully redefine our state, to let it include all relevant information. We performed several experiments in order to define the best values for the parameters, which are noted in the corresponding sections. When performing the final experiments, we saw that the method is successful in gaining good rewards and learning the correct actions. Hence, we will no longer look at the Q-learning algorithm, but only focus on deep reinforcement learning for the next cases.

In this chapter, we also answered sub-question 4a:

4. **What are the insights that we can obtain from our model?**
   (a) *How should the performance be evaluated for the first toy problem?*

We concluded that we are able to perform simulations while the method is still learning, to gain insights into the performance of our method. In this simulation, we calculate the holding and backorder costs, according to our reward function. With a fully learned policy, we are able to run the simulation again and look into the costs per time step. This way, we can see where the costs come from and easily compare them with a benchmark. For the next chapter, we will look into other ways to gain more insights into the method.

We now have successfully implemented deep reinforcement learning to the beer game. Looking back at the classification that was introduced in Chapter 2, we defined the beer game as follows:

T1: **4,S,**D,G|**I,G**|**N**,B|**O**,**F**,**N**|**C**|ACS,O
T2: **1**||**D**,P,**N**|P,**F**,N|**HB,N**

In this classification, the elements that differ from the case of CBC are given in bold. As can be seen, there are still various differences between the two cases. In the next chapter, we will try to close this gap by implementing the reinforcement learning method on another case from literature.

In the paper of Chaharsooghi et al. (2008), it is mentioned that future research in reinforcement learning should address the issue of having a non-linear network. As we will work towards the situation of CBC, which has a network of type *general*, we will first look into a less complicated non-linear network: a *divergent* network.

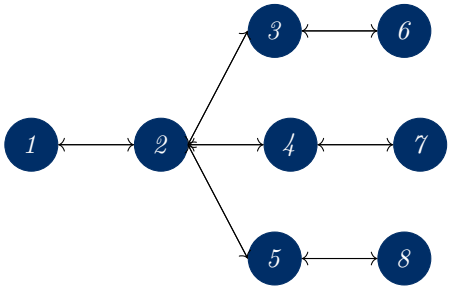
5. A Divergent Supply Chain

In this chapter, we will introduce a divergent supply chain to test our reinforcement learning method. We will first introduce the supply chain and its characteristics in Section 5.1. Section 5.2 covers the state variable of this supply chain, followed by the action variable in Section 5.3. The reward function and corresponding value function are given in Section 5.4 and Section 5.5 respectively. In Section 5.6, we describe how we adapted our current simulation to be able to implement the divergent supply chain. To benchmark the results of this divergent inventory system, we implemented a heuristic in Section 5.7. Section 5.8 covers the implementation and results of our deep reinforcement learning method. The conclusion can be found in Section 5.9.

5.1 Case description

In this section, we will discuss how we can apply our deep reinforcement learning method to a divergent supply chain. We choose for a divergent supply chain because it is not as complex as a general supply chain, but expands on the functionality of a linear one. With a divergent supply chain, our simulation has to be generalized and expanded. For example, orders can now be placed at a connected stock point, instead of only to one actor upstream. In Section 5.6, we will further elaborate on the changes we had to make to our simulation.

When we look at our concept matrix in Table 3.2, we can see that almost half of the papers consider a divergent network structure. Ideally, we are looking for a paper that lists the entire input and output data, just like Chaharsooghi et al. (2008) did. With this data, we can easily compare our method with the paper and adjust our method where necessary. Unfortunately, none of the listed papers included their input or output data, nor a detailed list of events that happen for every time step. The paper that is most suited for our research is the paper of Kunnumkal and Topaloglu (2011). As mentioned, this paper does not state their input or output data, but does clearly define specific settings of their divergent inventory system. They also mention the order of the activities that happen in every time period, but no detailed code of these events is given. Because this paper also uses another solving method (relaxation using Lagrange multipliers) then we do, we decided to only use their events list and parameters of the supply chain. Hence, we will define our own state- and action variable, and our own reward- and value function. Because of this, we also need to look for a benchmark to compare our result. We will elaborate on the benchmark in Section 5.7.



Where 1: Supplier, 2: Warehouse, 3, 4, 5: Retailer, 6, 7, 8: Customer

Figure 5.1: *Divergent supply chain.*

Kunnumkal and Topaloglu (2011) consider an inventory distribution system with one warehouse, denoted as ϕ, and three retailers ($r = 1, 2, 3$) and a planning horizon of 50 time periods ($t = 1, ...50$). The retailers face random demand and are supplied by the warehouse. The demand at retailer r at time period t is given by the random variable $d_{r,t}$. The demand of the warehouse is defined as $d_{\phi,t} = \sum d_{r,t}$, so that we can also speak of the demand at the warehouse at time period t. The warehouse replenishes its stock from an external supplier. All actors in this supply chain together are denoted as ($i = 1, 2, 3, 4, 5, 6, 7, 8$). A visualization of this supply chain can be seen in Figure 5.1.

A constant lead time of one time period is considered. Hence, the replenishment order shipped to a certain installation at a certain time period reaches its destination in the next time period. The following

events are defined per time period:

1. The warehouse places its replenishment order to the external supplier.
2. Considering the inventory positions at the warehouse and at the retailers, the warehouse ships the replenishment orders to the retailers.
3. The warehouse and retailers receive their replenishment shipment, shipped in the previous time period, from the external supplier and warehouse respectively.
4. The demand at the retailers is observed. The excess demand is backlogged by incurring backlogging costs. The warehouse and the retailers incur holding costs for the inventory that they carry to the next time period.

Because Kunnumkal and Topaloglu (2011) did not specify exactly what happens in these time steps, we had to make some assumptions. In the first event, all the stock points place their order to the upstream actor. The replenishment order of the retailers is not explicitly stated by Kunnumkal and Topaloglu (2011), but we assume that this happens this event, as the warehouse also places its order. The orders are placed simultaneously, so the warehouse does not know what the retailers order at that moment. The order sizes will be decided by the deep reinforcement learning method.

In the second event, the orders are shipped from the external supplier and the warehouse to the receiving stock points. In case the warehouse does not have enough inventory to fulfill all the orders, the orders will be fulfilled with respect to the inventory position. How this exactly happens is not described in the paper, but it makes sense to fully complete the orders, starting from the retailer with the lowest inventory position. Hence, the retailer with the lowest inventory position will be fulfilled first. If there is still inventory left in the warehouse, this will go to the retailer with the second lowest inventory position. In general, orders that cannot be fulfilled completely, will be fulfilled as far as possible. Remaining orders at the warehouse are not backlogged.

In event three, the stock points receive their shipment from the previous time period. This event uses the same logic as Event 2 of the beer game, which can be found in Algorithm 4.

In the fourth and last event, the demand at the retailers is observed. If the demand is larger than the inventory of the retailer, the remaining order quantity is backlogged.

In their base case, Kunnumkal and Topaloglu (2011) define three identical retailers. This means that the retailers all follow the same demand distribution. The holding and backlogging costs are $hc_{\phi,t} = 0.6$, $hc_{r,t} = 1$ and $bc_{r,t} = 19$ for all r and t. They assume that the demand quantity at retailer r at time period t has a Poisson distribution with mean $\alpha_{i,t}$. This alpha is generated from the uniform distribution in the interval $[5, 15]$ for every time step for every retailer.

The objective is to minimize the total expected holding and backorder costs of the warehouse and retailers. The decision variable is defined as the order quantity that will be decided for every stock point in every time period. Kunnumkal and Topaloglu (2011) formulate the problem as a dynamic program. Because the dynamic program is hard to solve due to its high dimensional state space, they relax the constraints by associating Lagrange multipliers with them. We will not go into further detail of their solution approach, as this is outside the scope of this research. Based on Chaharsooghi et al. (2008), we define the following variables and minimization formula:

$$minimize \sum_{t=1}^{n} (hc_{\phi,t} * h_{\phi,t} + \sum_{r=1}^{3} (hc_{r,t} * h_{r,t} + bc_{r,t} * b_{r,t})) \tag{5.1}$$

Where:

$$h_{\phi,t} = \begin{cases} IP_{\phi,t}, & \text{if } IP_{\phi,t} > 0 \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

$$h_{r,t} = \begin{cases} IP_{r,t}, & \text{if } IP_{r,t} > 0 \\ 0, & \text{otherwise} \end{cases} \tag{5.3}$$

$$b_{r,t} = \begin{cases} |IP_{r,t}|, & \text{if } IP_{r,t} \leq 0 \\ 0, & \text{otherwise} \end{cases} \tag{5.4}$$

Every time step, a decision on the order quantity has to be made. This results in the following transition to the next time step:

$$IP_{i,t+1} = IP_{i,t} + O_{i,j,t} - T_{i,j,t} \tag{5.5}$$

Where $IP_{i,t}$ represents the inventory position of stock point i in time step t, $(i = 2, 3, 4, 5)$

$O_{i,j,t}$ represents the ordering size of stock point i to the upstream level j, $(i = 2, 3, 4, 5, 6, 7, 8; j = 1, 2, 3, 4, 5)$

$T_{i,j,t}$ represents the distribution amount of level i to the downstream level j, $(i = 1, 2, 3, 4, 5; j = 2, 3, 4, 5, 6, 7, 8)$

With this information, we can now classify the divergent inventory system of Kunnumkal and Topaloglu (2011). According to the classification method of De Kok et al. (2018) and Van Santen (2019), we define the following classification:

```
T1: 2,D,D,G|I,G|P,B|O,F,N|C|ACS,O
T2: 1||D,P,N|P,O,N|HB,N
```

Now that we have classified the divergent inventory system, we will elaborate on the specific aspects of this system, including the state variable, reward function and value function.

5.2 State variable

For the divergent supply chain, we use an adaption of the state variable that we have defined in the deep reinforcement learning section of the beer game. Because we want to pass all the information that is available to the neural network, we define the following state vector:

$$S = [ti, tb, h_i, b_i, o_i] \tag{5.6}$$

In this case, ti again defines the total inventory in the system and tb defines the total number of backorders in the system. $h_i, (i = 2, 3, 4, 5)$ and $b_i, (i = 3, 4, 5)$ define the inventory and backorders per stock point respectively. Because unfulfilled demand for the warehouse does not result in backorders, we only use b_i for the retailers. o_i is the number of items that is arriving in the next time step for stock point i. As can be seen, this variable is no longer dependent on t, because we have a fixed lead time of 1. For this reason, we have removed tr_i from the state vector, as the total amount in transit is always equal to the total amount arriving next time period, when using a lead time of 1.

The values of these state variables are, once again, normalized to a range of $[-1, 1]$, to keep the values close together. We want the interval of the state variables before normalizing to have a range that is wide enough to ensure we do not limit our method in its observations, but also small enough that the neural network observes these states. The lower bound is 0 for every variable because we do not have negative orders, backorders, or shipments. We have experimented with several values for the upper bound and saw that these values can greatly impact the result of the method. Setting a too large or too small upper

bound, can cause the method to perform up to 50% worse. After our experiments, we used the upper bounds that are denoted in Table 5.1.

Table 5.1: *Upper bound of the state variables for the divergent supply chain.*

Variable	Upper bound
ti	1000
tb	450
h_i	250
b_i, o_i	150

5.3 Action variable

Just like in the beer game, for every time step, the order quantity has to be decided for every stock point. The demand is still unknown at the moment of ordering. We have to determine the complete size of the order. The action space for this case is formally defined as:

$$O_S = [O_{2,1,S}, O_{3,2,S}, O_{4,2,S}, O_{5,2,S}] \tag{5.7}$$

In this equation, the first subscripted number denotes the stock point that will order, while the second subscripted number denotes the stock point upstream that will receive the order. The action is state dependent, hence the subscripted S.

For our PPO algorithm, we will once again normalize our action space, and therefore have to define an upper bound for the action space. Because we want to encourage our agent to find the correct actions, and not limit it, we choose an upper bound of 300 for $O_{2,1,S}$ and 75 for the other actions. This should be high enough, without limiting the environment, because the highest mean of the Poisson distribution is 15. According to the Poisson distribution, this mean would result in a demand lower than 21 with a 95% confidence. The demand at the warehouse will be $(21 * 3 =)$ 63 in this case. Therefore, we think this upper bound is suitable.

5.4 Reward function

The objective of this case is to minimize the total holding and backorder costs in the supply chain. We will continue to use the PPO algorithm and therefore adapt our reward function from the beer game. The reward function is, therefore, determined as follows:

$$r(t) = hc_{\phi,t} * h_{\phi,t} + \sum_{r=1}^{3} [hc_{r,t} * h_{r,t} + bc_{r,t} * b_{r,t}] \tag{5.8}$$

In this equation, the holding and backorder costs for every location are included. The inventory costs for the warehouse are \$0.6 for every item and it does not have any backorders. The retailers have holding costs of \$1 per item and \$19 for every backorder.

5.5 Value function

The value function is used to denote the total expected costs over the time horizon. This function is correlated with the reward function. The value function is, once again, based on the value function of

Bellman, defined in Equation 3.2:

$$V(s) = \max_{a}(r(t) + \gamma \mathbb{E}[V(s`|s, a)]) \tag{5.9}$$

Where $r(t)$ is the reward function of Equation 5.4. Although Kunnumkal and Topaloglu (2011) do not consider discounting, we will include discounting in our model. Because we train our deep reinforcement learning on an infinite horizon, we do consider a discount rate of 0.99

5.6 Adapting the simulation

To use this case in our simulation, several things had to be adjusted. First of all, we had to make sure our simulation executed the events in the sequence that was defined by Kunnumkal and Topaloglu (2011). As mentioned earlier, the paper of Kunnumkal and Topaloglu (2011) did not specify exactly what happens in these events, hence we made some assumptions. To make sure our simulation worked with these events, only the first and second event had to be adjusted, as they were different from the events of the beer game. The adapted pseudo-code of these events can be found in Appendix G.

Next to that, we had to generalize our way of looking at orders and backorders. These can now be placed between any stock points with defined connections, instead of only directly upstream. We also added the compatibility with the Poisson demand distribution. Hence, the environment has become more general. The code of the inventory environment can be found in Appendix A.

In the beer game case, we were able to validate the simulation step by step, using the input and output data. Because we do not have that data for this case, but only the distributions, we cannot validate the simulation model the same way as we did for the beer game. Therefore, we used our visualization to debug the code. We used the visualization to see everything that happens during the events. This way, it is easy to detect inconsistencies in the simulation and to follow the shipments through the supply chain.

The last thing we had to add to the simulation is a warm-up period. Because the paper of Kunnumkal and Topaloglu (2011) did not state an initial inventory position, we decided to implement a warm-up period, to reach a state that is representative for a steady state. In other words, we remove the initialization and look only at the part where the simulation behaves as a steady running system. Hence, we start with an inventory system with no inventory, backorders or items in transit. To be safe, we chose for a warm-up period of 25 time steps. As mentioned earlier, the horizon of Kunnumkal and Topaloglu (2011) is 50 time periods. Therefore, we simulate for 75 periods, but remove the first 25 time steps from the result. This warm-up period is only used when evaluating the performance of the method. In the learning phase, it is not necessary to use a warm-up period, as the method should learn the value of all states.

5.7 Implementing a benchmark

We were hoping to use the paper of Kunnumkal and Topaloglu (2011) as a benchmark for our method. Unfortunately, when doing some test runs, we quickly saw that the results they mentioned in their paper, which are around 2700, were impossible for us to achieve. Because their events were not clearly stated and they used a different solution approach, we were not able to reproduce these results, nor able to see where this difference is coming from. Therefore, we decided to implement another benchmark. As we already discussed in Section 3.3, the most used benchmark is a base-stock policy. A base-stock policy determines the order-up-to level for every stock point. Hence, every time period, the stock point checks if its inventory position (inventory plus in transit minus backorders) is below the order-up-to level. If it is, it orders the amount needed to meet the order-up-to level.

In order to determine the base-stock levels for the divergent inventory system, we use the heuristic of Rong et al. (2017). This paper describes two heuristics that are able to successfully determine the near-optimal base-stock levels. We will use their decomposition-aggregation (DA) heuristic, as this heuristic is easier to implement, while being only slightly less accurate.

In the decomposition-aggregation heuristic, we initially decompose the divergent network into serial systems. For these decomposed networks, the heuristic of Shang and Song (2003) is used. The heuristic of Shang and Song can only be applied to linear supply chains, but is able to obtain near-optimal local base-stock levels for all locations in the linear system.

The second step of the DA heuristic is aggregation. Backorder matching is used to aggregate the linear supply chains back into the distribution network. The backorder matching procedure sets the local base-stock level of a given location in the divergent network so that its expected backorder level is equal to the sum of the expected backorder levels in all of its counterparts in the decomposed linear systems (Rong et al., 2017).

After using this heuristic, we found the base-stock levels [124, 30, 30, 30]. Hence, the base-stock level for the warehouse should be 124, while the base-stock level for the retailers is 30. The calculations for the DA heuristic can be found in Appendix H. Please note that we have used a lead time of 2 in these calculations instead of the lead time of 1 which was previously mentioned. This is due to the fact that Rong et al. (2017) use a continuous review, while our system uses a periodic review. In this case, demand per lead time is measured with $R + L$ instead of L. As this periodic review is 1 time period, this results in a total demand during lead time of $(R + L = 1 + 1 =)$ 2, which we will use in the heuristic (Silver, Pyke, & Thomas, 2016).

We can now perform our simulation using the base-stock levels that we found with the heuristic. We simulate for 75 periods and remove the first 25 periods due to the warm-up length. This simulation is replicated 1000. With these base-stock levels, we yield the average costs of 4059 for the 50 time steps.

5.8 Implementing Deep Reinforcement Learning

In the case of the beer game, we used the deep reinforcement learning method with the PPO algorithm. We saw that the method was able to yield good results and that it was able to use the full state and action size of the problem. In this section we will apply the deep reinforcement learning method to the divergent case.

To apply the method to a new case, only a few things have to be adjusted, being the previously described state variable, action variable, reward function and value function. We do not change any of the parameters of the PPO algorithm, except for the number of iterations, which is increased to 30.000. This way, one full run of the deep reinforcement learning method takes about 3.5 hours. Once again, we run the simulation every 100 iterations in order to evaluate the performance of the method. Every simulation is repeated 100 times, of which the average result is used. In this simulation, we use the warm-up period, as mentioned earlier. Hence, the simulation is performed for 75 time periods, of which the first 25 periods are removed from the results. In total, the whole method is run 10 times. The result of the deep reinforcement learning method can be seen in Figure 5.2. In this figure, we have removed the outliers, meaning that for every iteration, the highest and lowest scored rewards are removed, so 8 replications remain. The costs that we gained using the heuristic of Rong et al. (2017) are used as benchmark.

As can be seen in this figure, the method is able to correctly learn the needed actions quickly, as the costs decrease vastly over the first iterations. After only 30.000 iterations, which takes about 30 minutes, the

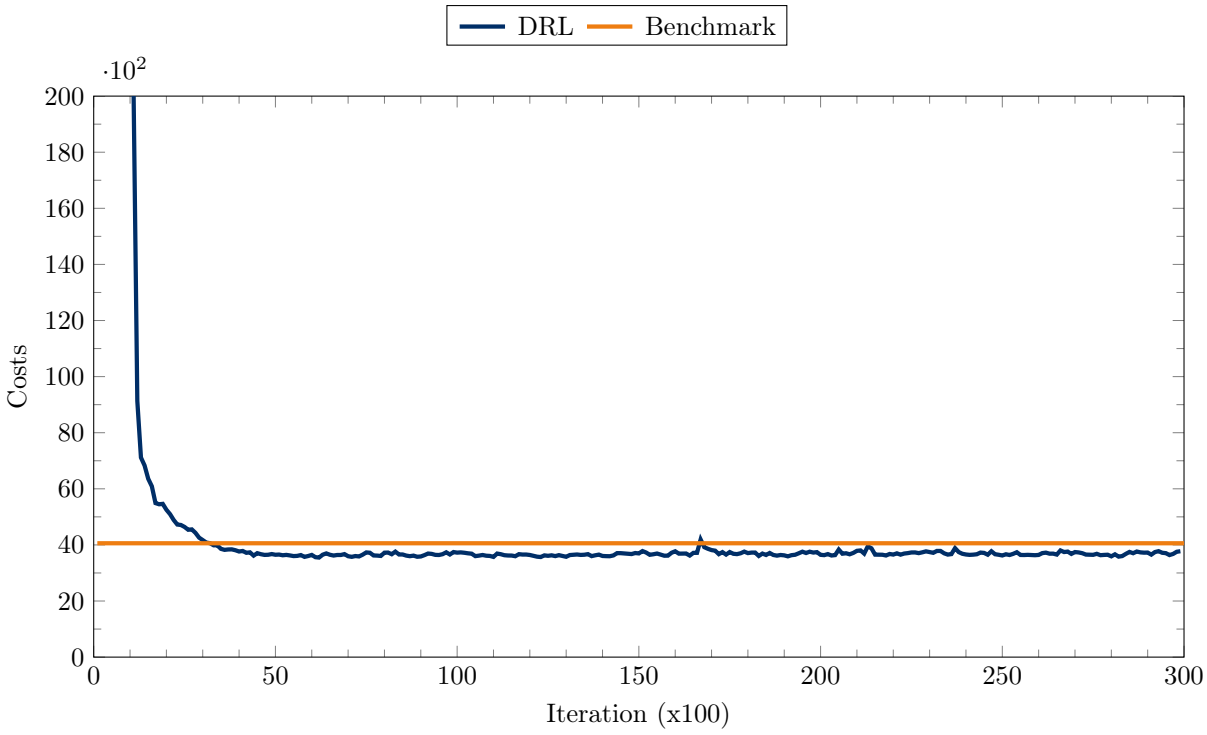


Figure 5.2: *Results of the PPO algorithm on the divergent supply chain. Lower is better.*

method is able to beat the benchmark. The final value of the costs is 3724, which means that the method is able to beat the benchmark, which is impressive. To gain more insights in the decisions that the deep reinforcement learning method makes, we will look how the actions of the method are dependent on the state. To do this, we use the trained network and feed the different states as input to this network. The output of the network is the action that it will take, hence, the quantity of the order it will place upstream. When we visualize these states and actions in a heatmap, we can see how the actions are coherent to the states. Because we use a two-dimensional heatmap, we can only vary two different variables in the state vector at a time, so the other values in the state vector get a fixed value. This value is based on the average value of the particular state variable, which we determined using a simulation. With this two-dimensional heatmap, it is not possibly to quickly see the relations between all the different states, yet we can get insights in some important connections.

In Figure 5.3, we can see such a heatmap. This heatmap shows the cohesion of the inventory in the warehouse with the number of items in transit for the warehouse. We can see that the order quantity gets lower as the inventory of the warehouse gets higher. The order quantity only slightly depends on the number of items in transit. When the number of items in transit is low, the order quantity is high. This order quantity slightly decreases when the items in transit increase. However, when the number of items in transit passes 50, the order quantity tends to increase again slightly. Although this result might not be as expected, we think this happens because the neural network does not observe these states often. The number of items in transit is equal to the order quantity of the previous period. In the heatmap, we can see that the order quantity is never higher than 65, for these state values. Hence, the number of items in transit will never be higher than 65. It could still be that another combination of state values does result in a order quantity, but this is a sign that the neural network does not obeserve these states often. All in all, the heatmap shows that the method was able to make connections between the state and actions that do make sense, but also has some actions that make less sense.

We decided to also take a look at the retailers and their state-action connections, which can be found in

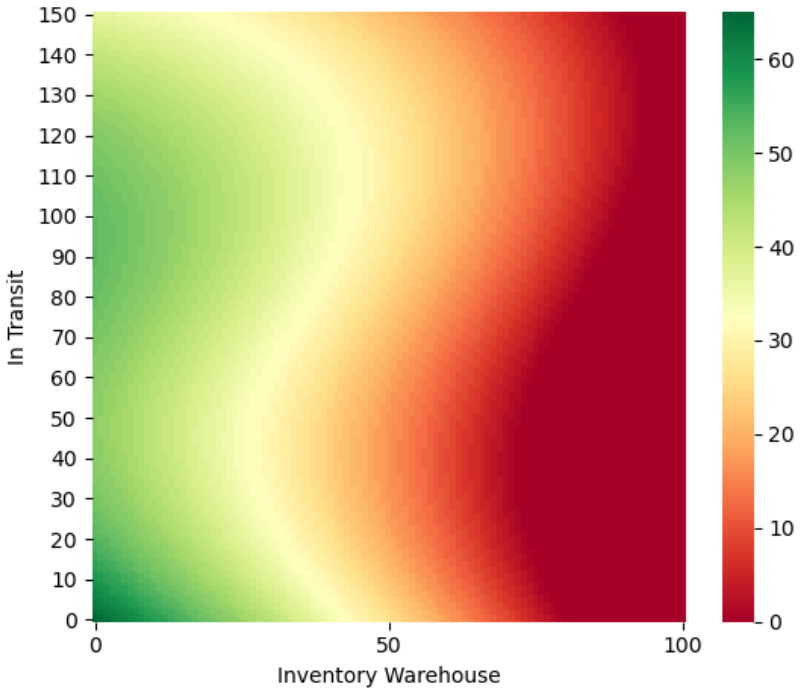


Figure 5.3: *Actions of the warehouse.*

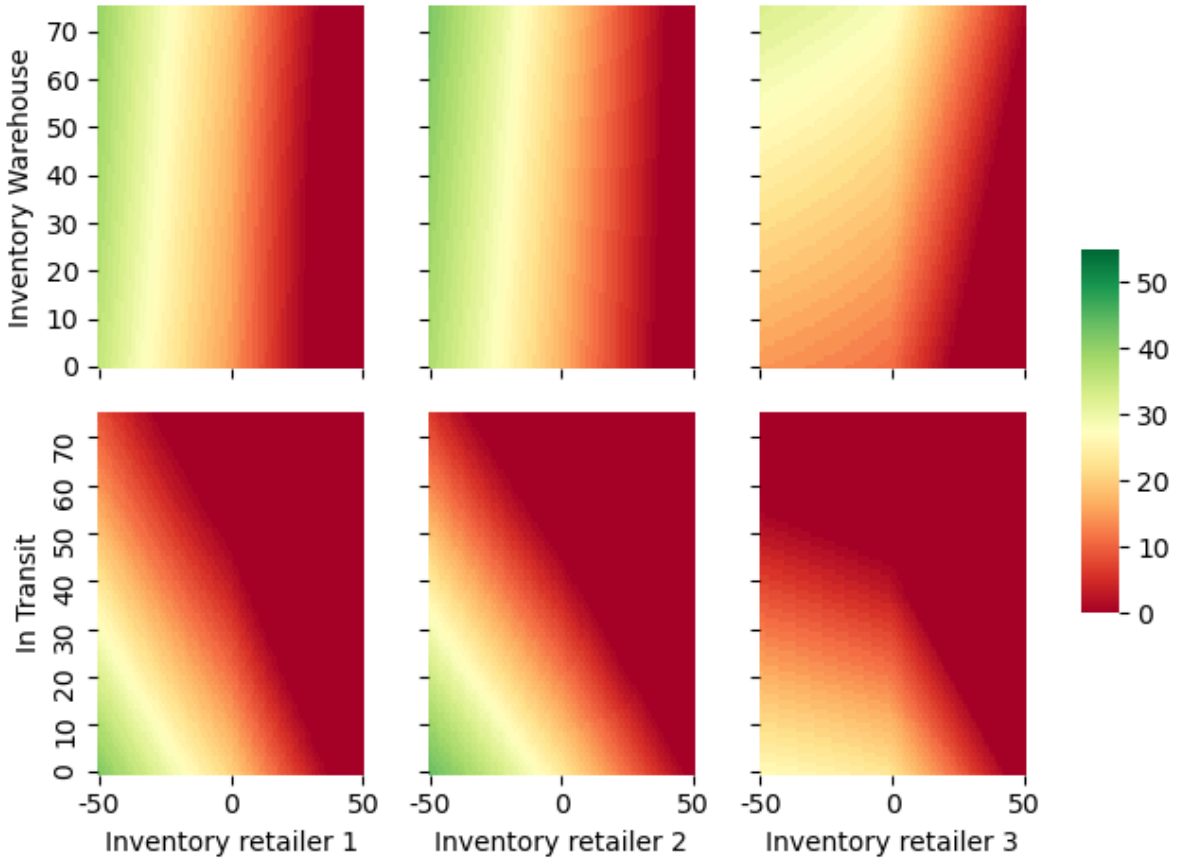


Figure 5.4: *Actions of the retailers.*

Figure 5.4. For these heatmaps, we looked at the different retailers and the relation of their inventory to both the inventory of the warehouse as the number of items in transit. As we can see, the actions are not really dependent on the inventory of the warehouse. This does make sense, as the number of items in the warehouse does not directly affect the inventory position of the retailer. The connection between the number of items in transit and the inventory of the retailer is clearly visible. The method shows that it does not order when the number of items in transit is large. Also, the current inventory of the retailers impacts the order quantity. Especially for retailer 1 and 2, the order quantity quickly increases when the retailers have backorders. Retailer 3 does not show this quick increase, but still shows some increase.

We can conclude that the deep reinforcement learning method can easily be applied to other cases, without alterations to the algorithm. There might be still some improvement possible by tuning the parameters of the algorithm. However, this is a non-trivial and time-consuming task (Gijsbrechts et al., 2019). Therefore, we decided to keep the current parameters. With the current settings, the method is already able to beat a base-stock level heuristic. Hence, the main finding is that the deep reinforcement learning method can easily be applied to other cases, while it is also able to yield good results.

5.9 Conclusion

In this chapter, we have successfully implemented our deep reinforcement learning method on a divergent inventory system from literature, in order to answer sub-question 3b:

3. **How can we build a reinforcement learning method to optimize the inventory management at CBC?**
   (b) *How can we expand the model to reflect another clearly defined problem from literature?*

We used the case described by Kunnumkal and Topaloglu (2011) to define the divergent inventory system. We elaborated on the specific elements of this case in Sections 5.1 to 5.5. We made the corresponding alterations to our existing simulation, in such a way that it was able to define the case of Kunnumkal and Topaloglu (2011), as described in Section 5.6. In Section 5.7, we used a heuristic of Rong et al. (2017) to determine the base-stock policy levels, which served as a benchmark for our deep reinforcement learning method.

When implementing the deep reinforcement learning method in Section 5.8, we discovered that no alterations to the PPO algorithm are needed to get it to work for the divergent inventory system. However, we did need to define a new state and action vector and noticed that these vectors can have a great impact on the performance of the method. Varying the values of these vectors can result in a performance that is up to 50% worse. The current values are gained by several experiments. The method is able to quickly learn the correct actions and even beats the benchmark, which is impressive, considering the near-optimality of base-stock heuristics. When we take a closer look at the actions that the method takes, it is shown that, apart from some exceptions, it is able to learn the relations between certain states and actions. We can conclude that it was easy to implement the PPO algorithm to another case. The state and action variables do have to be adjusted, as they are case specific. However, setting these values is not trivial and we concluded that setting a too large or too small upper bound, can cause the method to perform up to 50% worse.

In this chapter, we also covered sub-question 4b:

4. **What are the insights that we can obtain from our model?**
   (b) *How should the performance be evaluated for the second toy problem?*

In this case, we have, once again, used a simulation to evaluate the performance of the method. Next to that, we were able to create heatmaps to gain more insights into our method. These heatmaps show the action that the method will take, in relation to two state variables. Because we can only use two variables, it is hard to show how all the variables are related to each other.

Looking back at the classification that was introduced in Chapter 2, we defined the divergent system as follows:

T1: 2,**D,**D,G|**I**,G|**P**,B|**O**,**F**,**N**|**C**|ACS,O
T2: **1**||**D**,P,**N**|P,O,N|**HB,N**

In this classification, the elements that differ from the case of CBC are, once again, given in bold. Although we have closed the gap between the beer game case and the case of CBC, there are still various differences. In the next chapter, we will discuss these differences, as the next chapter will describe the implementation of deep reinforcement learning to the case of CBC.

6. The CardBoard Company

In this chapter, we will elaborate on the case of the CardBoard Company. We will first take a look at the real-life case of CBC and make the translation to the simulation in Section 6.1. Section 6.2 covers the state variable of this supply chain, followed by the action variable in Section 6.3. The reward function and corresponding value function are given in Section 6.4 and Section 6.5 respectively. To be able to correctly compare the results of the deep reinforcement learning method, we discuss the benchmark in Section 6.6. Section 6.7 covers the implementation and results of our deep reinforcement learning method. The practical implications of the method are discussed in Section 6.8. The conclusion can be found in Section 6.9.

6.1 Case description

In Chapter 2, we have introduced the real-life case of CBC. We have gathered all the relevant information of the system and classified it according to the typology of De Kok et al. (2018) and Van Santen (2019). In this section, we will take a closer look at this classification and determine which elements we are able to implement in our deep reinforcement learning method. The classification of CBC is as follows, where the letters in bold indicate that this feature has not yet been implemented in the method:

T1: 2,**G**,D,G|**F**,C|**C**,B|**s**,**O**,**R**|**SC**|ACS,O
T2: **n**||**M**,P,**I**|P,O,N|**N,B**

In the next subsections, we elaborate on the most important features of the CardBoard Company and we discuss how the gap between the case of CBC and the simulation will be closed. For every feature, we decide if this will be implemented in the simulation.

Products

CBC produces various different types of cardboards. As mentioned in Chapter 2, in total, 281 unique products can be produced. Not every plant can produce every product type and there are types that can be produced in multiple corrugated plants, which results in a total of 415 unique product type – corrugated plant combinations.

For our simulation, we decided to only look at one product type at a time. If the simulation were to include multiple products, this would heavily impact the number of actions, as the order policy is determined for every product. This would, therefore, also impact the running time of our deep reinforcement learning method. To determine the order policy for multiple products, the method can be run multiple times.

However, this simplification also impacts other aspects of the CBC case. We mentioned earlier that every stock point in the supply chain has a bounded capacity. However, because this capacity is large, and, therefore, only interesting when looking at multiple products, we consider the bounded capacity out of scope.

When a certain product type is not available, customers of CBC can choose for substitution of the product. This way, the customer can choose for another, more expensive product to be delivered instead. Because substitution is only possible when multiple products are considered in the simulation, we also consider substitution to be out of scope. Hence, products that are not available will not be substituted by another product but will be backlogged instead.

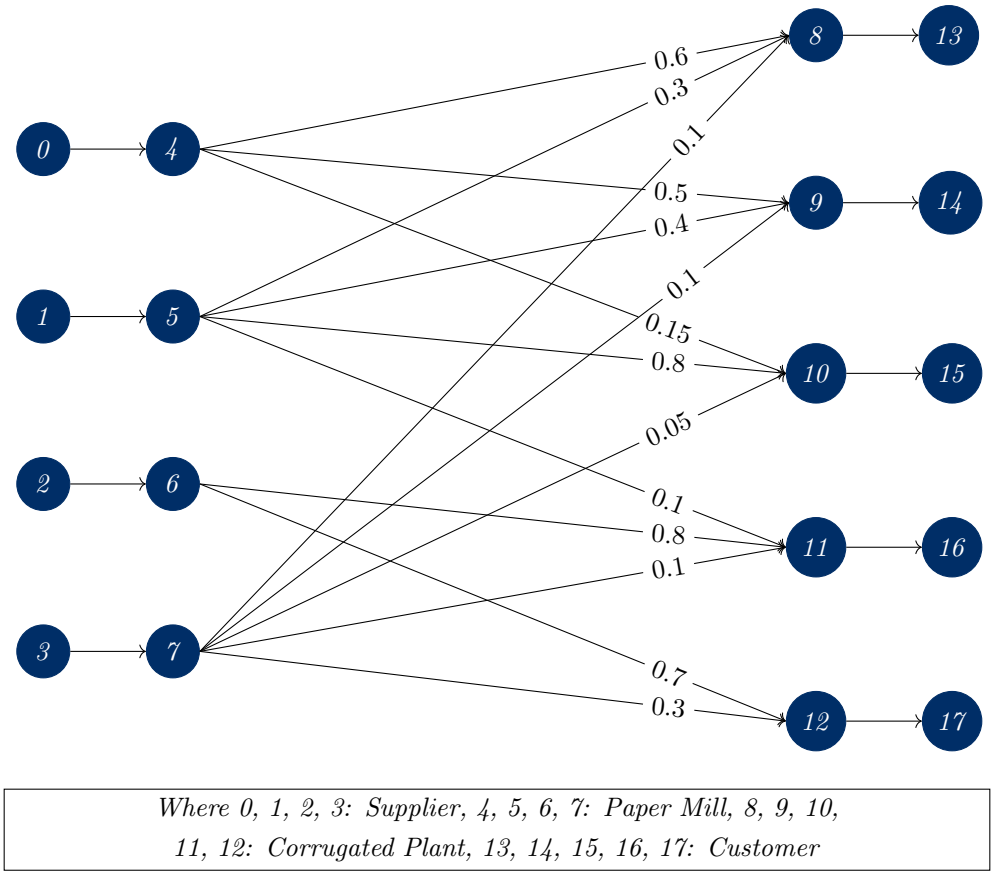


Figure 6.1: *An overview of the supply chain of CBC, along with the probabilities of the connections between the paper mills and corrugated plants.*

Connections

The CardBoard Company has an inventory system with multiple connections between the stock points and is therefore classified as a general inventory system. In our simulation model, we will add the possibility to add multiple connections per corrugated plant. There are two different options to let the corrugated plants cope with multiple connections where they can place their order.

The first option is to let the deep reinforcement learning method decide at which upstream stock point the plant will place its order. This way, we have to define an action for every possible connection, which would result in 18 actions in the case of CBC. The second option, which is also denoted as relative rationing, is to assign a certain probability to every connection. This probability represents the chance that the plant will order at this connection. This way, we have to define an action for every stock point, which results in 9 actions, and the probabilities of the connections should sum op to 1 for every plant.

Relative rationing does add extra randomness to the environment, which could be difficult to learn for the deep reinforcement learning method. However, because we rather want to limit the amount of actions that the deep reinforcement learning method has to learn, we decided to implement relative rationing. Relative rationing is also used by the CardBoard Company itself, when running simulations and evaluations of their supply chain. Figure 6.1 denotes paper mills and corrugated plants of CBC, along with the probabilities of the connections.

In Chapter 2, we mentioned that we assume a constant lead time, as the observed lead times did not vary more than 30 minutes on lengths of multiple hours. In real-life, this lead time is different for every

connection, but, for simplicity, we will assume the lead time to be equal for every connection. We will assume a lead time of 1 time period.

The last simplification concerns the routing flexibility. It is possible that a product type is not on stock on a certain corrugated plant but is available at another plant. In the real-life situation, the product can then be delivered from one corrugated plant to the other, which is described as routing flexibility. These connections only exist between a few plants that are closely located at each other. In our simulation, we will not take this routing flexibility into account.

Demand

The historic demand of the CardBoard Company from last year was extracted from the ERP system. However, when taking a closer look at the demand, we noticed that it was lumpy demand. This means that the variability in both the demand timing and demand quantity is high, and, therefore, hard to predict. For simplicity, we decided to use a Poisson distribution with $\lambda = 15$ to generate the demand. In the real-world situation of CBC, paper mills deliver not only to the corrugated plants, but also have other customers that can order directly at the paper mills, which is called intermediate demand. For now, we consider this intermediate demand to be out of scope, so customer demand can only occur at the plants.

As mentioned in Chapter 2, CBC considers a Minimum Order Quantity, hence, if an order is placed, it should at least be higher than the MOQ. In our simulation, we do not consider a MOQ.

Inventory rationing has to be used whenever two or more orders can only be partially fulfilled. In our simulation, we will, just like in the previous case, look at the inventory position of the corrugated plants. The plant with the lowest inventory position will be fulfilled first. We do not need inventory rationing for the customers, as we only generate one order per time period.

Objective

In Chapter 2, we mentioned that CBC wants to minimize their costs, while obtaining a target service level of 98%. The real holding and backorder costs of CBC are unknown, and, therefore, we will have to define our own costs. We will use the same holding and backorder costs as defined by Kunnumkal and Topaloglu (2011) in the divergent supply chain. Hence, the holding costs ($hc_{r,t}$) will be \$0.6 for the paper mills and \$1 for the corrugated plants. The backorder costs ($bc_{r,t}$) will be \$19 for the corrugated plants. There are no backorder costs for the paper mills. For now, we will not impose the target service level, but rather focus on the minimization of costs.

The decision variable is defined as the order quantity that will be decided for every stock point in every time period. Based on the two previous cases, we define the following variables and minimization formula:

$$minimize \sum_{t=1}^{n} (\sum_{r=4}^{12} [hc_{r,t} * h_{r,t} + bc_{r,t} * b_{r,t}]) \tag{6.1}$$

Where:

$$h_{r,t} = \begin{cases} IP_{r,t}, & \text{if } IP_{r,t} > 0 \\ 0, & \text{otherwise} \end{cases} \tag{6.2}$$

$$b_{r,t} = \begin{cases} |IP_{r,t}|, & \text{if } IP_{r,t} \leq 0 \\ 0, & \text{otherwise} \end{cases} \tag{6.3}$$

Every time step, a decision on the order quantity has to be made. This results in the following transition to the next time step:

$$IP_{i,t+1} = IP_{i,t} + O_{i,j,t} - T_{i,j,t} \tag{6.4}$$

Where $IP_{i,t}$ represents the inventory position of stock point i in time step t, $(i = 4, ..., 12)$

$O_{i,j,t}$ represents the ordering size of stock point i to the upstream level j, $(i = 4, ..., 12; j = 0, ..., 7)$

$T_{i,j,t}$ represents the distribution amount of level i to the downstream level j, $(i = 0, ..., 7; j = 4, ..., 12)$

n represents the planning horizon of 50 time periods. In these time periods, we will use the same order of events as listed in the beer game case of Chaharsooghi et al. (2008).

New classification

With these assumptions and simplifications, we end up with the following classification. This classification denotes all the features that will be implemented in the simulation. The bold values indicate the differences with the real-life case of CBC.

T1: 2,G,D,G|**I**,C|**P**,B|**O**,**F**,**N**|**C**|ACS,O
T2: **1**||**D**,P,**N**|P,O,N|HB,**N**

As can be seen, there are still quite some features that differ from the situation of CBC. We think it is important to experiment with a simplified general network first, to see if the deep reinforcement learning method is able to learn. If this is the case, future research could focus on extending the simulation in such a way that all features of the case of CBC can be covered.

6.2 State variable

To define the state variable for CBC, we, once again use an adaption of the state variables that we have defined in the previous cases. Because we want to pass all the relevant information that is available to the neural network, we define the following state vector:

$$S = [ti, tb, h_i, b_{ij}, o_{ij}] \tag{6.5}$$

To adapt the state vector to a general inventory system, we changed the last three variables in this vector. These variables are now denoted per connection, instead of per stock point. Hence, the backorders and the number of items arriving in the next time step are now dependent on both i and j, where i is the source of the connection, while j is the destination.

The values of these state variables are, as opposed to the previous cases, normalized to a range of $[0, 1]$. We have changed this interval because we want the output of an untrained network in the first state to have 0 as mean. When starting with a mean of 0, we make sure that the actions, do not have to be clipped in the beginning, as the actions can lie in the interval of $[-1, 1]$. In the previous sections, we have arbitrarily determined the upper bound for the state and action vectors. Because we have concluded that these upper bound values have a great impact on the performance of the method, we have decided to perform experiments to determine these upper bounds. In Section 6.7, we will elaborate on these experiments.

6.3 Action variable

The deep reinforcement learning method will decide on the order quantity for every stock point. The demand is still unknown at the moment of ordering and we have to determine the complete size of the

order. The action space for this case if formally defined as:

$$O_S = [O_{4,0,S}, O_{5,1,S}, O_{6,2,S}, O_{7,3,S}, O_{8,j,S}, O_{9,j,S}, O_{10,j,S}, O_{11,j,S}, O_{12,j,S}] \tag{6.6}$$

In this equation, the first subscripted number denotes the stock point that will place the order, while the second subscripted number denotes the upstream stock point that will receive the order. If a j is used, this means that the upstream actor can vary, according to the probabilities of the connections, as given in Figure 6.1.

We will once again normalize our action space to the interval of $[-1, 1]$, and therefore have to define an upper bound for the action space. We will run experiments in Section 6.7 to define suitable values for the upper bound.

6.4 Reward function

The objective of this case is to minimize the total holding and backorder costs in the supply chain. The reward function is based on the two previous cases and denoted as follows:

$$r(t) = \sum_{r=4}^{12} [hc_{r,t} * h_{r,t} + bc_{r,t} * b_{r,t}] \tag{6.7}$$

In this equation, the holding and backorder costs for every location are included. The inventory costs for the paper mills are \$0.6 for every item and the backorder costs are zero. The retailers have holding costs of \$1 per item and \$19 for every backorder.

6.5 Value function

The value function is used to denote the total expected costs over the time horizon. This function is correlated with the reward function. The value function is, once again, based on the value function of Bellman, defined in Equation 3.2:

$$V(s) = \max_a (r(t) + \gamma \mathbb{E}[V(s`|s, a)]) \tag{6.8}$$

Where $r(t)$ is the reward function of Equation 6.7. Just like in the previous cases, our gamma is set to 0.99. The value function will be estimated by the PPO algorithm.

6.6 Benchmark

To compare the result of our deep reinforcement learning method, we are looking for a suitable benchmark. For the divergent supply chain, we were able to implement a heuristic of Rong et al. (2017) that computes the parameters for a base-stock policy. However, this heuristic is only suitable for divergent supply chains, so we are not able to use this benchmark for the case of CBC.

We do, however, have a benchmark from the real-life case of CBC that we can reconstruct. As mentioned in Chapter 2, the CardBoard Company tries to achieve a fill rate of 98% for their customers. They want to achieve this by also enforcing the fill rate of 98% for the connections between the paper mills and corrugated plants. With this information, we are able to set up a benchmark. We do this by manually tuning the base-stock parameters in such a way that for every connection, the fill rate of 98% is met. We measure this fill rate by running the simulation for 50 time periods and replicate this simulation 500 times. With this method, we yield the following base stock parameters: [82, 100, 64, 83, 35, 35, 35, 35, 35], which results in the average total costs of 10467.

6.7 Implementing Deep Reinforcement Learning

In the two previous cases, we used deep reinforcement learning with the PPO algorithm. The method was able to yield good results with this algorithm. However, we did notice that the performance of the algorithm was heavily dependent on the definition of the state and action vectors. In this section, we will apply our method to the case of the CardBoard Company. We will first define several experiments to determine the best suitable values for upper bound of the state and action vectors. When we have determined these values, we can run the experiments to measure the performance of the method.

In the previous chapter, we have concluded that a good definition of both the state and action vector is critical for a correctly performing method. Because we have already determined base-stock levels in the previous section, we have an indication for the values of the action vector. We define three different possible value combinations that we will experiment on: $[300, 150]$, $[150, 75]$ and $[150, 50]$. The first value denotes the upper bound of the actions for the paper mills, while the last value denotes the upper bound for the corrugated plants.

The upper bound values of the state vector should be set such that they are high enough to ensure we do not limit our method in its observations, but also small enough such that the neural network observes these states. With the use of simulations, we have defined three possible combinations for the state values: $[1000, 1000, 500]$, $[500, 500, 250]$ and $[250, 250, 150]$. In these combinations, the first value denotes the upper bound for the inventory for all stock points. The second value concerns the backorders for the paper mills, while the last value concerns the backorders for the corrugated plants. The upper bounds for the total inventory and total backorders are defined by the sum of all upper bounds for the inventory and backorders respectively. The upper bounds for the items in transit is dependent on the action values. For now, we have not considered specific values for individual connections, yet, this might be interesting to take into account for future experiments. Table 6.1 defines the unique experiments that will be performed.

Table 6.1: *Experiments for defining the upper bounds of the state and action vector for the CBC case.*

Experiments	Action space	State space
1	[300, 75]	[1000, 1000, 500]
2	[300, 75]	[500, 500, 250]
3	[300, 75]	[250, 250, 150]
4	[150, 75]	[1000, 1000, 500]
5	[150, 75]	[500, 500, 250]
6	[150, 75]	[250, 250, 150]
7	[150, 50]	[1000, 1000, 500]
8	[150, 50]	[500, 500, 250]
9	[150, 50]	[250, 250, 150]

We did not change any of the parameters of the PPO algorithm, aside from the new scaling, mentioned in Section 6.2. We run the deep reinforcement learning method for 30.000 iterations, which now takes about 4.2 hours, because of the increased output layer. This time, we run the simulation every 500 iterations in order to evaluate the performance. Every simulation takes 100 time steps, of which the first 50 time steps are removed because of the warm-up period. The simulation is repeated 100 times, of which the average is used. In total, we run the experiments for two replications. This way, we can quickly determine which settings performs best. The results are visualized in Figure 6.2.

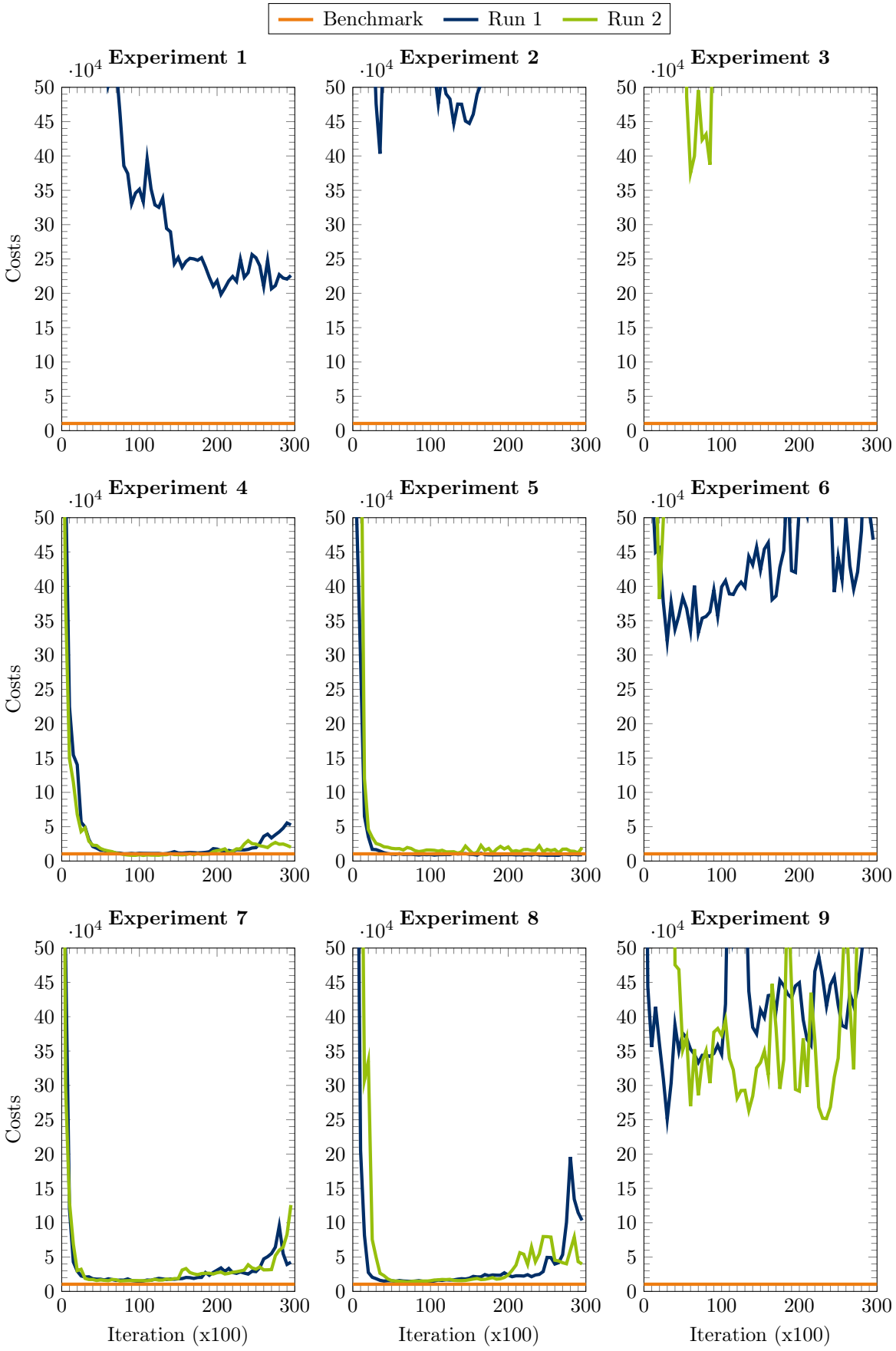


Figure 6.2: *Results of the PPO algorithm per experiment. Lower is better.*

As can be seen, the results vary greatly between the different experiments, but also vary between the different runs. Sometimes, the method performs so bad, that the results are outside the scope of our figure. We can quickly see that there are four experiments that were able to approximate the benchmark, being Experiments 4, 5, 7 and 8. Experiment 5 performs the best, as this experiment yields steady results and does still improve over the iterations. We, therefore, choose to continue with Experiment 5 and completed 10 runs in total with these settings. Figure 6.3 shows the performance of Experiment 5 on 10 runs. As can be seen, the results of these runs are varying. Five out of ten runs are performing well and lie close to the benchmark. However, the other five runs yield poor results and are not able to learn correct order quantities. For the remaining analysis, we will first focus on the performance of the runs that are learning correctly. Thereafter, we will gain more insights into the other runs that are not learning.

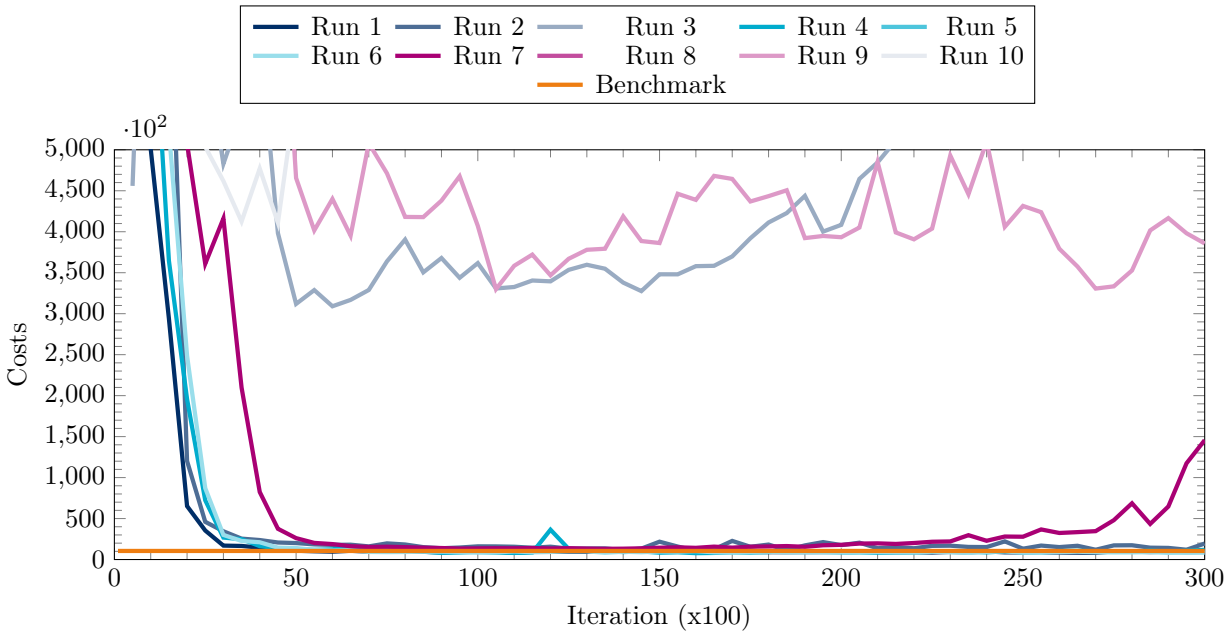


Figure 6.3: *Results of the PPO algorithm on the case of CBC. Lower is better.*

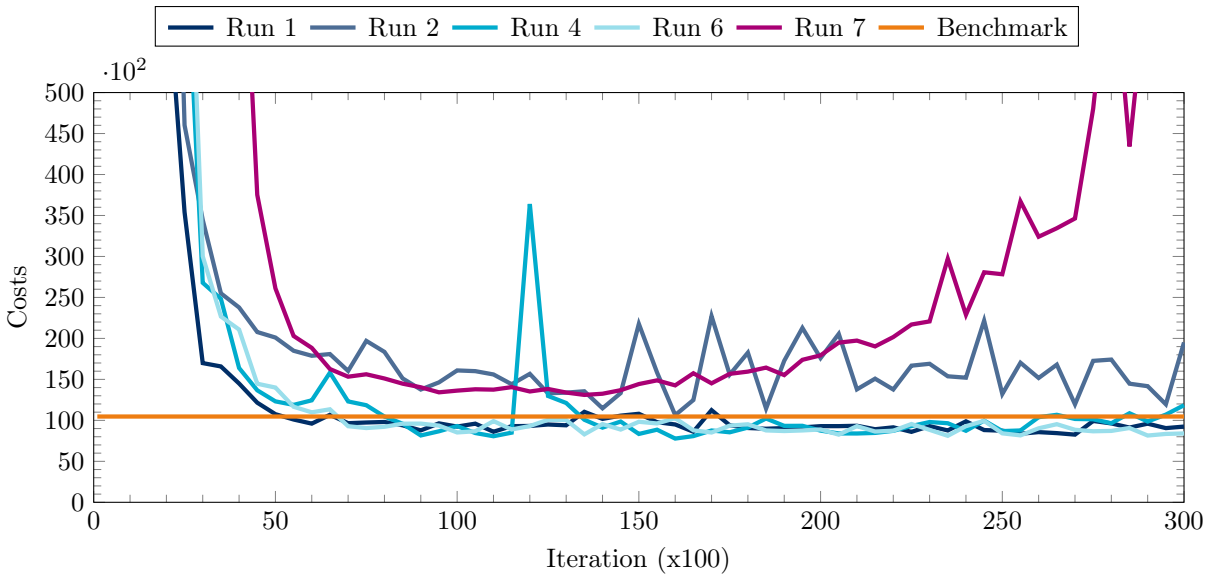


Figure 6.4: *Results of the PPO algorithm on the case of CBC of the 5 best runs. Lower is better.*

Figure 6.4 zooms in on the five runs that are learning correctly. Although these runs are also still varying in their results, we can see that all of them approach the benchmark, and sometimes even score better than the benchmark. In the divergent case, we created heatmaps to gain more insights into the actions of the method. In the case of the CardBoard Company, we are not able to create these heatmaps, as we have too many state variables that depend on each other. To gain more insights into the origins of the rewards and how these compare to the benchmark, we will visualize the costs per time step. Next to that, we will compare the achieved fill rates of the benchmark and our deep reinforcement learning method.

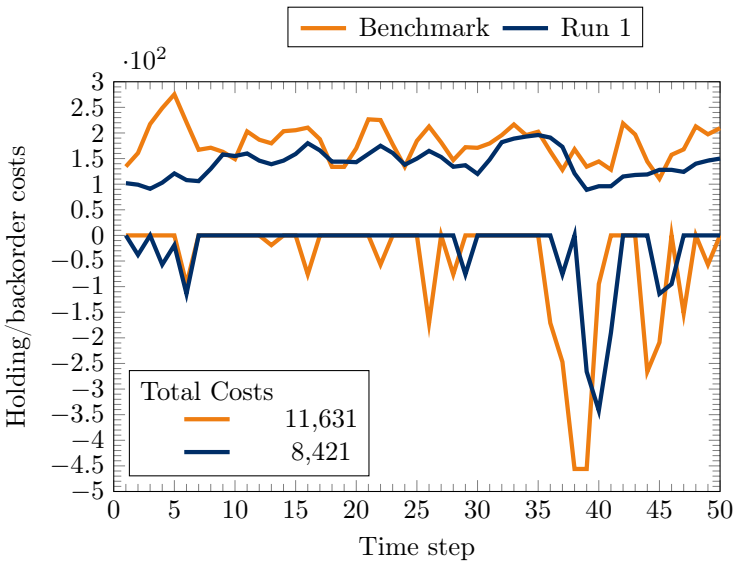


Figure 6.5: *Costs over time for the case of CBC. Closer to zero is better.*

For every run of our deep reinforcement learning method, we use the fully trained network to run a simulation for 150 periods. We then remove the first and last 50 periods, to account for the warm-up period and possible end effects. Figure 6.5 shows the results of Run 1. As we can see, this run is scoring significantly better than the benchmark. The holding costs of our method are somewhat lower than the benchmark, but the biggest difference can be found in the backorder costs. Because, most of the time, both the holding and backorder costs are lower than the benchmark for our method, this shows that the method has better distributed its inventory over the different stock points.

Next to that, we also want insights into the average fill rate that the method was able to yield. We have calculated the fill rate using the same simulation. Figure 6.6 shows the average fill rate of Run 1 for both the benchmark as our deep reinforcement learning method. While we did not impose the 98% fill rate as a constraint in our model, the method is still meeting this requirement for almost every corrugated plant. We can see that the method did learn that it is not necessary to achieve a 98% fill rate for every stock point, but only for the corrugated plants, as they are directly delivering to the customer. Overall, we can say that the method is able to yield good results, but is not able to yield these results for all different runs. The method is, therefore, unstable and not yet fitted to use in a real-life situation.

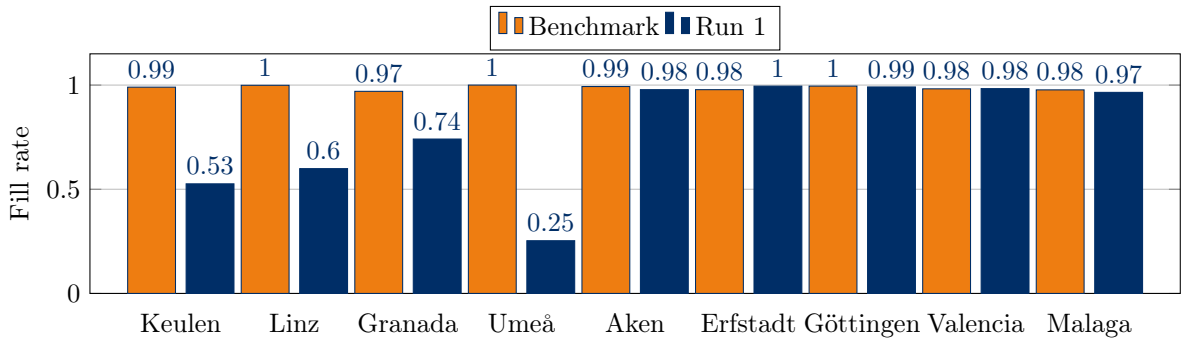
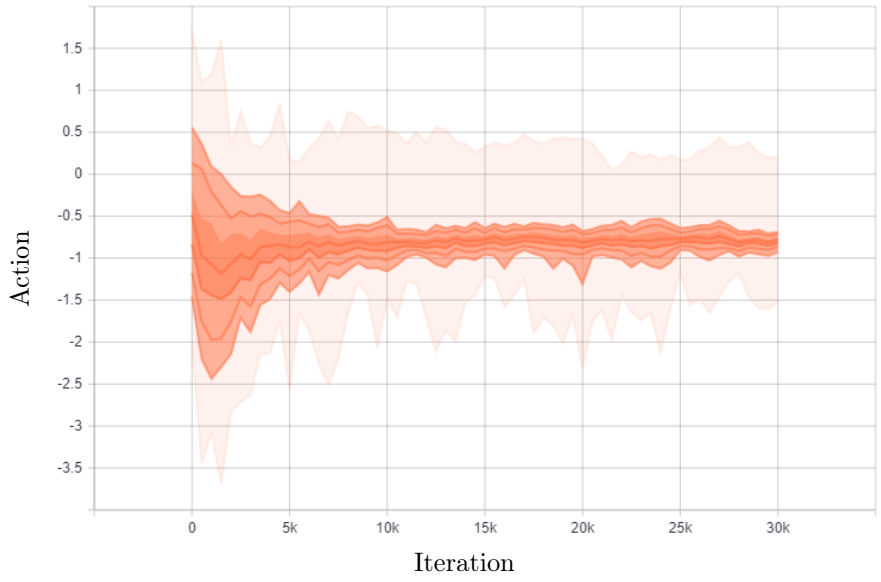
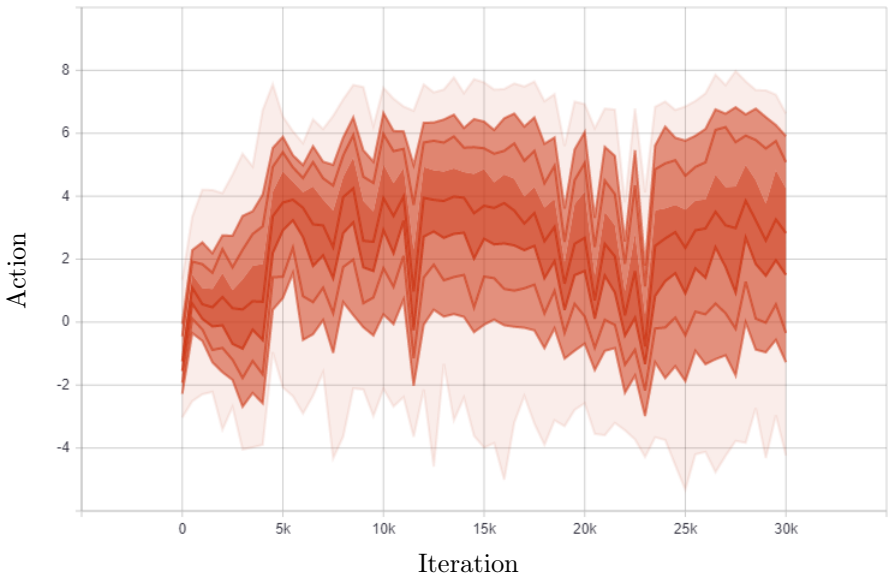


Figure 6.6: *Average fill rates for the stock points of CBC.*

It is hard to say why the method is not performing well for all runs. However, if we take a look at the actions that the method takes in different runs, we can see a clear difference in runs that are yielding good results compared to runs that are not performing well. As mentioned in Section 6.3, actions are scaled to the interval $[-1, 1]$. If the method takes actions outside this interval, these actions are clipped. Hence, choosing either -1 or -3 will result in the same action. Ideally, the method is able to learn this interval and chooses actions within these bounds. However, we noticed that this was not the case for every run. Figure 6.7 shows the distribution of scaled actions that the deep reinforcement learning method takes, for runs 1 and 10. We can see that for run 1, the method is able to learn that it should place its actions in the interval of $[-1, 1]$. For run 10, the method does not place its actions inside this interval. This is also the case for all other runs that are not performing well. Future research could, therefore, look into these actions and experiment with different intervals.



(a) Run 1



(b) Run 10

Figure 6.7: *Distribution of the scaled actions.*

6.8 Practical implications

Now that we have implemented the deep reinforcement learning method on the case of the CardBoard Company, we can elaborate on the practical implications of this method. We discuss how this method can be implemented at CBC and how ORTEC can use the method.

As mentioned in Chapter 4, we explained that we have implemented the deep reinforcement learning method on an infinite horizon. Hence, our method proposes a policy that is not time dependent, but only depends on the states. This results in a policy that is comparable with a base-stock policy, as the base-stock policy is also dependent on the state of the stock point. When we have run the method, we end up with a trained neural network. This neural network can directly be used CBC. Every time when CBC wants to place its orders, it could input its current state of the inventory system to the neural network, which will then output the best order quantities, that it has determined during training. However, determining the current state of the inventory system can be difficult. Therefore, it could be interesting to link the neural network to the ERP system of CBC, such that the neural network can automatically extract the needed information about the current state.

In order to apply the deep reinforcement learning method to other customers, two aspects should be considered. First, it is important that all relevant information of the customers inventory system is gathered. This can be done using the classification of De Kok et al. (2018) and Van Santen (2019). When every feature of this classification is covered, we have enough information to translate the real-life situation to the simulation. The second important aspect is to tune the state and action variables. This can be done by defining several experiments, just like we did in the previous section. When the experiments determined the best suitable state and actions variable, the method can be run and the network can be trained. This network can then, just like described in the previous paragraph, be used to obtain the fitting order quantities for every state.

The method is built in an iterative way. We started by applying the deep reinforcement learning method to two toy problems and, thereafter, applied the method to the case of CBC. This way, we can ensure the generalization of the method. However, we did not yet implement logic for all different elements of the classification. Therefore, it could be that the simulation first has to be extended before it is able to reflect the situation of a certain inventory system.

6.9 Conclusion

In this chapter we have implemented our deep reinforcement learning on the third and final case: the case of the CardBoard Company. With this implementation, there are several research questions that we can answer.

3. **How can we build a reinforcement learning method to optimize the inventory management at CBC?**
   (c) *How can we expand the model to reflect the situation of CBC?*

In Section 6.1, we have reviewed the classification of the CardBoard Company and covered the aspects that were not yet implemented in the simulation. We have made several assumptions and simplifications in order to determine the scope of our simulation. Eventually, we have implemented the following inventory system in our simulation, according to the classification of De Kok et al. (2018) and Van Santen (2019):

T1: 2,G,D,G|**I**,C|**P**,B|**O**,**F**,**N**|**C**|ACS,O
T2: **1**||**D**,P,**N**|P,O,N|HB,**N**

In this classification, the bold values denote the differences with the real-life situation of the CardBoard Company.

Next to that, we had to determine suitable action and state variables. This is done by running several experiments, using predefined action and state intervals. With these experiments, we have managed to find a setting that is able to yield good results. With this setting, we have performed more experiments in order to answer our following research question:

4. **What are the insights that we can obtain from our model?**
   (c) *How should the performance be evaluated for CBC?*
   (d) *How well does the method perform compared to the current method and other relevant methods?*

In Section 6.6, we concluded that heuristics for general inventory systems are scarce and we decided to use the current method of CBC as benchmark. Because we made several assumptions to the simulation, we had to reconstruct this method, by setting base-stock parameters in such a way that all connections satisfy a 98% fill rate. We know that this method is not optimal, but it still can be used as a fair benchmark.

We ran the final experiments for the deep reinforcement learning method in Section 6.7. We evaluated the performance every 500 iterations, by running a simulation of 50 time periods. We were not able to create heatmaps for the case of CBC, as there are too many state variables that depend on each other. We ran a simulation with our method and the benchmark, to see where the costs per time step come from. We also calculate the average fill rate per stock point. However, we also noticed that the performance of the method differs heavily per run. Moreover, we noticed that some runs do not seem to learn at all. These runs start with bad results and do not improve over time. We took a closer look at the actions of the method, and noticed that it is not able to learn the correct interval of the scaled actions. Therefore, we think that future research can focus on experimenting with this interval.

In Section 6.8, we have elaborated on the practical implications, to answer the following research question:

5. **How can ORTEC use this reinforcement learning method?**
   (a) *How can the new method be implemented at CBC?*
   (b) *How can the reinforcement learning method be generalized to be used for inventory systems of other customers?*

With the deep reinforcement learning method, we have trained a neural network. This neural network can directly be used CBC. Every time when CBC wants to place its orders, it could input its current state of the inventory system to the neural network, which will then output the best order quantities, that it has determined during training.

The method is already applied at three cases and has been built in an iterative way. This way, we can ensure the generalization of the method. If ORTEC wants to use the deep reinforcement learning method at other customers, two aspects are important:

- The inventory systems should be fully classified. This way, ORTEC can make sure that the inventory system can be implemented in the simulation
- The state and action variables should be defined for every unique inventory system, and have to be tuned.

7. Conclusion and Recommendations

In this chapter, we discuss the most important findings and implications of this research. In Section 7.1, we discuss the most important findings, the contributions to literature, and the current limitations. Section 7.2 covers the recommendations for ORTEC. Suggestions for further research are given in Section 7.3.

7.1 Conclusion

This research was initiated by ORTEC. ORTEC has experience with traditional solving methods for multi-echelon inventory management, but is interested in the applicability of reinforcement learning on this subject. A suitable case for this research is the case of the CardBoard Company. This company has a multi-echelon inventory system and is looking to minimize their holding and backorder costs. Therefore, we formulated the following research question:

In what way, and to what degree, can a reinforcement learning method be best applied to the multi-echelon inventory system of the CardBoard Company, and how can this model be generalized?

To answer this research question, we started by looking into the current situation of the CardBoard Company. We described all relevant aspects of the supply chain and denoted them using the classification concept of De Kok et al. (2018) and Van Santen (2019), with which we were able to capture all real-life elements.

We have built a reinforcement learning method in an iterative way. We started with two toy problems: the beer game and a divergent supply chain of Kunnumkal and Topaloglu (2011). For the beer game, we have reconstructed the simulation and the Q-learning method from the paper of Chaharsooghi et al. (2008). We concluded that the method of Chaharsooghi et al. (2008) was not learning the correct values of the states, but only gained their results using random actions. This was possible because they defined a small action space, where the impact of the chosen actions was only limited. We also concluded that the tabular Q-learning method was not scalable, as it already reached the memory limits of our computer. Therefore, we decided to implement a deep reinforcement learning method with the Proximal Policy Optimization algorithm of Schulman et al. (2017). This method uses a neural network to estimate the value function, and can, therefore, also be used on larger problems. We performed several experiments with the deep reinforcement learning method and concluded that is was able to learn, as it yields better results than the method of Chaharsooghi et al. (2008).

Because of the successful implementation of the deep reinforcement learning method on the beer game, we also applied this method to a divergent inventory system, defined by Kunnumkal and Topaloglu (2011). We were able to apply the method without any modifications of the parameters of the algorithm. We did notice that the definition of state and action vector were important for the performance of the algorithm. In order to compare our results, we implemented the heuristic of Rong et al. (2017), which determines the near optimal base-stock parameters for divergent supply chains. After running several experiments, we see that our deep reinforcement learning method is able to beat this benchmark after only 30.000 iterations, which takes about 30 minutes.

After two successful implementations of our deep reinforcement learning method, we applied our method to a simplified case of the CardBoard Company. As a benchmark, we have reconstructed the current method of CBC, by determining the base-stock parameters in such a way that a fill rate of 98% is yielded for every connection. As we concluded that the state and action vector are critical for performance of the method, we defined nine experiments that varied on the action and state space. Of these nine experiments, we choose one setting to further experiment on. With this setting, we can see that the

method succeeds in yielding better results than the benchmark for some runs. However, the method turns out to be unstable and is not able to learn the correct actions for five out of ten runs. We have looked into the actions of the runs that are not performing well, and noticed that the method is not able to learn the interval of the action space. Therefore, we believe that more experiments regarding the action space are necessary and can still improve the result. The final results of the deep reinforcement learning method on the three different cases can be found in Table 7.1. For the case of the CardBoard Company, we have denoted both the best run and the worst run, as these runs are far apart.

Table 7.1: *Total costs of the DRL method and the corresponding benchmarks. Lower is better.*

Case	DRL	Benchmark
Beer game	**2,726**	3,259
Divergent	**3,724**	4,059
CBC	8,402 - 1,252,400	**10,467**

We have compared the deep reinforcement learning to various benchmarks. For the beer game, we were able to compare our result with the result of the paper. For the divergent case, we have implemented the base-stock heuristic of Rong et al. (2017) in order to compare our method. Heuristics for determining base-stock parameters in general inventory systems are hard to find, which is why we decided to compare our method with the reconstructed current method of CBC. We have evaluated the performance using a simulation and compare the total costs gained in the horizon.

We have used to different methods to gain more insight in the decisions of the neural network. At first, we have applied the neural network alongside a benchmark to the same simulation instance to compare their performance side by side. This way, we were able to see the actions that the method took every time step and the corresponding costs. Moreover, we have created a heatmap in which the order quantity is shown, dependent on two state variables. This way, we can see the dynamic of the action and see if these actions seem logical choice.

Our deep reinforcement learning method can be implemented at CBC by using the trained neural network. Every time when CBC wants to place its orders, it could input its current state of the inventory system to the neural network, which will then output the best order quantities, that it has determined during training. However, determining the current state of the inventory system can be difficult. Therefore, it could be interesting to link the neural network to the ERP system of CBC, such that the neural network can automatically extract the needed information about the current state.

In order to apply the deep reinforcement learning method to other customers, ORTEC should take two things into account. It is important that all relevant information of the customers inventory system is gathered. A way to ensure all relevant information is captured, the classification of De Kok et al. (2018) and Van Santen (2019) can be used. The second important aspect is to tune the state and action variables for every specific case. When the best suitable state and actions variable are found, the method can be run and the network can be trained. This network can then be used to obtain the fitting order quantities for every state.

7.1.1 Scientific contribution

This research contributes to the fields of deep reinforcement learning and inventory management in three ways. To the best of our knowledge, this research is the first example that has successfully applied a

neural network with a continuous action space to a multi-echelon inventory system. With the use of the Proximal Policy Optimization algorithm, we are able to beat the benchmark on both a linear and a divergent system. To accomplish this, we have created a general environment in which various supply chains can be modeled and solved. This environment can also be used in future research and extended in order to simulate even more important aspects of inventory systems. At last, we have applied deep reinforcement learning to a general inventory system, a case that has not been considered before in deep reinforcement learning. However, it must be noted that the performance of the method is still varying and needs to be improved in order to actually use it as solution method.

7.1.2 Limitations

Despite the promising results, this research and our method comes with limitations. We discuss limitations of the data and the method used in this research.

The foremost limitation of this research is the difficulty with which the performance of the deep reinforcement learning method can be compared with the actual performance of the CardBoard Company. This limitation is self-imposed, as we have decided to make simplifications to the case of CBC. These simplifications made the case easier to implement in our simulation. The simulation is only an approximation and does not fully capture all details of the real-life situation of CBC. Therefore, this research is not fully able to show precise improvements made possible by the deep reinforcement learning method.

One of these simplifications concerns the demand. While we do have realized demand data available, we have decided not to use this and use a Poisson distribution instead. Because this demand turned out to be lumpy for various products, this assumption can have a large impact on the performance of the method.

At last, our research covered several inventory systems and various benchmarks. However, for the general inventory system, we have used the current method of CBC as a benchmark, while we know that this method is not optimal. Therefore, we do not know how well the deep reinforcement learning method performs in comparison with other, more eloquent techniques.

7.2 Recommendations

This research has shown that deep reinforcement learning can be a suitable technique to solve various inventory problems. If ORTEC wants to further develop the method in order to use it for complex customer cases, we have the following recommendations to ORTEC:

- Do not start with using deep reinforcement learning as main solution to customers yet. Rather use the method on the side, to validate how the method performs in comparison with other solution approaches. It is also recommended to implement various benchmarks in the simulation itself, to quickly see the performance of the deep reinforcement learning method in various settings.

- Before offering the solution to customers, first focus on the explainability of the method. By default, deep reinforcement learning is sometimes considered to be a 'black box' system, because it can be unclear why the method chose a certain action. However, this does not have to be the case. We have tried to gain insights in the decisions that our method makes and various researches have focused on the explainability of neural networks and deep reinforcement learning. We recommend ORTEC to also focus on the explainability of the method.

- Look for ways to reduce the complexity of the environment and the deep reinforcement learning method. Instead of deciding on the full order size, the deep reinforcement learning method could,

for example, be used to determine the optimal base-stock parameters. This can make the action space smaller and less dependent on the various states. This would reduce the agent's training time and increase the possibility of finding feasible solutions for more complex environments.

- Keep a close eye on developments in the deep reinforcement learning field. The number of applications to Operation Research and Inventory Management is limited, but still gains interest quickly. For deep reinforcement learning, various algorithms are publicly available. These algorithms often have been implemented in several Python packages. Investigating what the best algorithm is for which type of problem can increase the performance of the deep reinforcement learning method. We recommend to focus on the algorithms that allow a continuous action space, as these are considered to be more scalable and, therefore, suitable for general inventory problems.

7.3 Future research

This thesis introduces a deep reinforcement learning method that is able to perform well on various inventory systems. However, the usage of this method also raised new questions, that resulted in suggestions for further research.

We noticed that the state and action vectors do have a large effect on the performance. Defining these vectors is not trivial and case specific. Therefore, we recommend to look for ways to automate and optimize this process.

Next to that, we also saw that the action space interval is currently not correctly used in several runs. For future research, we recommend to experiment with different intervals in order to improve the method and make sure the results do not differ as much between runs as they do now.

We have now implemented the deep reinforcement learning method to a simplified version of the case of CBC. Therefore, our suggestion for future research is to further expand the simulation to let it reflect the real-life situation of CBC. This way, the method can also be used on other, more complex inventory systems. This is especially interesting for inventory systems that are hard to solve with more traditional methods, such as heuristics and mathematical models.

Next to that, it would be interesting to use the original demand data of CBC. This demand data is lumpy and, therefore, hard to predict. We are interested to see if deep reinforcement learning is still able to determine order quantities when the demand gets harder to predict and the stochasticity of the environment increases.

We have implemented the Proximal Policy Optimization algorithm, as it is the current state-of-the-art algorithm that is less sensitive to hyperparameters and, therefore, easier to apply to multiple cases. However, it could also be interesting to take a look at other DRL algorithms, as they might be a better fit for specific cases. Research has been done on comparing the algorithms on various games, but not yet on an inventory setting. Therefore, we recommend to implement various algorithms to the inventory environment, to compare their performance on this domain.

Bibliography

Achiam, J. (2018). Spinning Up in Deep Reinforcement Learning. Retrieved from https://github.com/openai/spinningup

Arnold, J., Chapman, S., & Clive, L. (2008). *Introduction to Materials Management*. Pearson Prentice Hall.

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, *34*(6), 26–38. doi:10.1109/MSP.2017.2743240

Axsäter, S. (2015). *Inventory Control*. International Series in Operations Research & Management Science. Springer International Publishing.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-dynamic Programming*. Anthropological Field Studies. Athena Scientific. Retrieved from https://books.google.nl/books?id=WxCCQgAACAAJ

Carlos, D. P. A., Jairo, R. M. T., & Aldo, F. A. (2008). Simulation-optimization using a reinforcement learning approach. *Proceedings - Winter Simulation Conference*, 1376–1383. doi:10.1109/WSC.2008.4736213

Chaharsooghi, S. K., Heydari, J., & Zegordi, S. H. (2008). A reinforcement learning model for supply chain ordering management: An application to the beer game. *Decision Support Systems*, *45*(4), 949–959. doi:10.1016/j.dss.2008.03.007

Chaudhary, V., Kulshrestha, R., & Routroy, S. (2018). State-of-the-art literature review on inventory models for perishable products. doi:10.1108/JAMR-09-2017-0091

Chen, F., Feng, Y., & Simchi-Levi, D. (2002). Uniform distribution of inventory positions in two-echelon periodic review systems with batch-ordering policies and interdependent demands. *European Journal of Operational Research*, *140*(3), 648–654. doi:10.1016/S0377-2217(01)00203-X

Chen, F., & Song, J.-S. (2001). Optimal Policies for Multiechelon Inventory Problems with Markov-Modulated Demand. *Operations Research*, *49*(2), 226–234. doi:10.1287/opre.49.2.226.13528

Chen, W., & Yang, H. (2019). A heuristic based on quadratic approximation for dual sourcing problem with general lead times and supply capacity uncertainty. *IISE Transactions*, *51*(9), 943–956. doi:10.1080/24725854.2018.1537532

Chopra, S., & Meindl, P. (2015). *Supply Chain Management: Strategy, Planning, and Operation, Global Edition* (6th). Pearson Education Limited.

Çimen, M., & Kirkbride, C. (2013). Approximate dynamic programming algorithms for multidimensional inventory optimization problems. *IFAC Proceedings Volumes (IFAC-PapersOnline)*, *46*(9), 2015–2020. doi:10.3182/20130619-3-RU-3018.00441

Çimen, M., & Kirkbride, C. (2017). Approximate dynamic programming algorithms for multidimensional flexible production-inventory problems. *International Journal of Production Research*, *55*(7), 2034–2050. doi:10.1080/00207543.2016.1264643

De Kok, A. G., Grob, C., Laumanns, M., Minner, S., Rambau, J., & Schade, K. (2018). A typology and literature review on stochastic multi-echelon inventory models. *European Journal of Operational Research*, *269*(3), 955–983. doi:10.1016/j.ejor.2018.02.047

Ganeshan, R. (1999). Managing supply chain inventories: A multiple retailer, one warehouse, multiple supplier model. *International Journal of Production Economics*, *59*(1), 341–354. doi:10.1016/S0925-5273(98)00115-7

Gemmink, M. (2019). The adoption of reinforcement learning in the logistics industry : A case study at a large international retailer, 143.

Geng, W., Qiu, M., & Zhao, X. (2010). An inventory system with single distributor and multiple retailers: Operating scenarios and performance comparison. *International Journal of Production Economics*, *128*(1), 434–444. doi:10.1016/j.ijpe.2010.08.002

Giannoccaro, I., & Pontrandolfo, P. (2002). Inventory management in supply chains: A reinforcement learning approach. *International Journal of Production Economics*, *78*(2), 153–161. doi:10.1016/S0925-5273(00)00156-0

Gijsbrechts, J., Boute, R. N., Van Mieghem, J. A., & Zhang, D. (2019). Can Deep Reinforcement Learning Improve Inventory Management? Performance and Implementation of Dual Sourcing-Mode Problems. *SSRN Electronic Journal*, (July). doi:10.2139/ssrn.3302881

Gumus, A. T., & Guneri, A. F. (2007). Multi-echelon inventory management in supply chains with uncertain demand and lead times: literature review from an operational research perspective. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, *221*. doi:10.1243/09544054JEM889

Gumus, A. T., & Guneri, A. F. (2009). A multi-echelon inventory management framework for stochastic and fuzzy supply chains. *Expert Systems with Applications*, *36*(3), 5565–5575. doi:10.1016/j.eswa.2008.06.082

Gumus, A. T., Guneri, A. F., & Ulengin, F. (2010). A new methodology for multi-echelon inventory management in stochastic and neuro-fuzzy environments. In *International journal of production economics* (Vol. 128, *1*, pp. 248–260). doi:10.1016/j.ijpe.2010.06.019

Habib, N. (2019). *Hands-On Q-Learning with Python: Practical Q-learning with OpenAI Gym, Keras, and TensorFlow*. Packt Publishing. Retrieved from https://books.google.nl/books?id=xxiUDwAAQBAJ

Hausman, W. H., & Erkip, N. K. (1994). Multi-Echelon vs. Single-Echelon Inventory Control Policies for Low-Demand Items. *Management Science*, *40*(5), 597–602. doi:10.1287/mnsc.40.5.597

Hayes, G. (2019). How to Install OpenAI Gym in a Windows Environment. Retrieved from https://towardsdatascience.com/how-to-install-openai-gym-in-a-windows-environment-338969e24d30

Iida, T. (2001). The infinite horizon non-stationary stochastic multi-echelon inventory problem and near-myopic policies. *European Journal of Operational Research*, *134*(3), 525–539. doi:10.1016/S0377-2217(00)00275-7

Jaakkola, T., Jordan, M. I., & Singh, S. P. (1994). On the Convergence of Stochastic Iterative Dynamic Programming Algorithms. *Neural Computation*, *6*(6), 1185–1201. doi:10.1162/neco.1994.6.6.1185

Jiang, C., & Sheng, Z. (2009). Case-based reinforcement learning for dynamic inventory control in a multi-agent supply-chain system. *Expert Systems with Applications*, *36*(3 PART 2), 6520–6526. doi:10.1016/j.eswa.2008.07.036

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, *4*(9), 237–285. doi:10.1613/jair.301

Kalchschmidt, M., Zotteri, G., & Verganti, R. (2003). Inventory management in a multi-echelon spare parts supply chain. *International Journal of Production Economics*, *81*(82), 397–413. doi:10.1016/S0925-5273(02)00284-0

Kenton, W. (2019). Overfitting. Retrieved from https://www.investopedia.com/terms/o/overfitting.asp

Kim, C. O., Jun, J., Baek, J.-G., Smith, R. L., & Kim, Y. D. (2005). Adaptive inventory control models for supply chain management. *International Journal of Advanced Manufacturing Technology*, *26*(9-10), 1184–1192. doi:10.1007/s00170-004-2069-8

Kim, C. O., Kwon, I. H., & Baek, J.-G. (2008). Asynchronous action-reward learning for nonstationary serial supply chain inventory control. *Applied Intelligence*, *28*(1), 1–16. doi:10.1007/s10489-007-0038-2

Kim, C. O., Kwon, I. H., & Kwak, C. (2010). Multi-agent based distributed inventory control model. *Expert Systems with Applications*, *37*(7), 5186–5191. doi:10.1016/j.eswa.2009.12.073

Kimbrough, S. O., Wu, D. J., & Zhong, F. (2002). Computers play the beer game: Can artificial agents manage supply chains? *Decision Support Systems*, *33*(3), 323–333. doi:10.1016/S0167-9236(02)00019-2

Krzyk, K. (2018). Coding Deep Learning For Beginners - Types of Machine Learning. Retrieved from https://www.experfy.com/blog/coding-deep-learning-for-beginners-types-of-machine-learning

Kunnumkal, S., & Topaloglu, H. (2011). Linear programming based decomposition methods for inventory distribution systems. *European Journal of Operational Research*, *211*(2), 282–297. doi:10.1016/j.ejor.2010.11.026

Kwak, C., Choi, J. S., Kim, C. O., & Kwon, I. H. (2009). Situation reactive approach to Vendor Managed Inventory problem. *Expert Systems with Applications*, *36*(5), 9039–9045. doi:10.1016/j.eswa.2008.12.018

Kwon, I. H., Kim, C. O., Jun, J., & Lee, J. H. (2008). Case-based myopic reinforcement learning for satisfying target service level in supply chain. *Expert Systems with Applications*, *35*(1-2), 389–397. doi:10.1016/j.eswa.2007.07.002

Lambrecht, M., Luyten, R., & Vander Eecken, J. (1985). Protective inventories and bottlenecks in production systems. *European Journal of Operational Research*, *22*(3), 319–328. doi:10.1016/0377-2217(85)90251-6

Lambrecht, M., Muchstadt, J., & Luyten, R. (1984). Protective stocks in multi-stage production systems. *International Journal of Production Research*, *22*(6), 1001–1025. doi:10.1080/00207548408942517

Li, J., Guo, P., & Zuo, Z. (2008). Inventory control model for mobile supply chain management. *Proceedings - The 2008 International Conference on Embedded Software and Systems Symposia, ICESS Symposia*, 459–463. doi:10.1109/ICESS.Symposia.2008.85

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... Wierstra, D. (2016). Continuous control with deep reinforcement learning. In *4th international conference on learning representations, iclr 2016 - conference track proceedings*. Retrieved from http://arxiv.org/abs/1509.02971

Lin, L.-J. (1991). Self-improvement Based On Reinforcement Learning, Planning and Teaching. *Machine Learning Proceedings 1991*, *321*, 323–327. doi:10.1016/b978-1-55860-200-7.50067-2

Littman, M. L. (2015). Reinforcement learning improves behaviour from evaluative feedback, 5–11. doi:10.1038/nature14540

Mes, M. R. K., & Rivera, A. P. (2017). Approximate Dynamic Programming by Practical Examples. In *International series in operations research and management science* (Vol. 248, *February*, pp. 63–101). doi:10.1007/978-3-319-47766-4{_}3

Minner, S., Diks, E. B., & De Kok, A. G. (2003). A two-echelon inventory system with supply lead time flexibility. *IIE Transactions (Institute of Industrial Engineers)*, *35*(2), 117–129. doi:10.1080/07408170304383

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., ... Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. *33rd International Conference on Machine Learning, ICML 2016*, *4*, 2850–2869. Retrieved from http://arxiv.org/abs/1602.01783

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. doi:10.1038/nature14236

Muccino, E. (2019). Scaling Reward Values for Improved Deep Reinforcement Learning. Retrieved from https://medium.com/mindboard/scaling-reward-values-for-improved-deep-reinforcement-learning-e9a89f89411d

Nahmias, S., & Smith, S. A. (1993). Mathematical Models of Retailer Inventory Systems: A Review. In *Perspectives in operations management* (pp. 249–278). doi:10.1007/978-1-4615-3166-1{_}14

Nahmias, S., & Smith, S. A. (1994). Optimizing Inventory Levels in a Two-Echelon Retailer System with Partial Lost Sales. *Management Science*, *40*(5), 582–596. doi:10.1287/mnsc.40.5.582

OpenAI. (2015). About OpenAI. Retrieved from https://openai.com/about/

Oroojlooyjadid, A., Nazari, M., Snyder, L., & Takáč, M. (2017). A Deep Q-Network for the Beer Game: A Deep Reinforcement Learning algorithm to Solve Inventory Optimization Problems. *Europe*, *79*(0), 21. Retrieved from http://arxiv.org/abs/1708.05924

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, d\textquotesingle Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Powell, W. B. (2011). *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Wiley.

Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A., & Dormann, N. (2019). Stable Baselines. \url{https://github.com/DLR-RM/stable-baselines3}. GitHub.

Rana, A. (2018). Introduction: Reinforcement Learning with OpenAI Gym. Retrieved from https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2

Rao, J. J., Ravulapati, K. K., & Das, T. K. (2003). A simulation-based approach to study stochastic inventory-planning games. *International Journal of Systems Science*, *34*(12-13), 717–730. doi:10.1080/00207720310001640755

Rao, U., Scheller-Wolf, A., & Tayur, S. (2000). Development of a Rapid-Response Supply Chain at Caterpillar. *Operations Research*, *48*(2), 189–204. doi:10.1287/opre.48.2.189.12380

Rau, H., Wu, M. Y., & Wee, H. M. (2003). Integrated inventory model for deteriorating items under a multi-echelon supply chain environment. *International Journal of Production Economics*, *86*(2), 155–168. doi:10.1016/S0925-5273(03)00048-3

Ravulapati, K. K., Rao, J., & Das, T. K. (2004). A reinforcement learning approach to stochastic business games. *IIE Transactions (Institute of Industrial Engineers)*, *36*(4), 373–385. doi:10.1080/07408170490278698

Robinson, S. (2004). *Simulation: The Practice of Model Development and Use* (2nd). Wiley.

Rong, Y., Atan, Z., & Snyder, L. V. (2017). Heuristics for Base-Stock Levels in Multi-Echelon Distribution Networks. *Production and Operations Management*, *26*(9), 1760–1777. doi:10.1111/poms.12717

Russell, S., & Norvig, P. (2009). *Artificial Intelligence: A Modern Approach* (3rd). USA: Prentice Hall Press.

Saitoh, F., & Utani, A. (2013). Coordinated rule acquisition of decision making on supply chain by exploitation-oriented reinforcement learning -beer game as an example-. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *8131 LNCS*, 537–544. doi:10.1007/978-3-642-40728-4{_}67

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms, 1–12. Retrieved from http://arxiv.org/abs/1707.06347

Shang, K. H., & Song, J. S. (2003). Newsvendor bounds and heuristic for optimal policies in serial supply chains. *Management Science*, *49*(5), 618–638. doi:10.1287/mnsc.49.5.618.15147

Shervais, S., & Shannon, T. T. (2001). Improving theoretically-optimal and quasi-optimal inventory and transportation policies using adaptive critic based approximate dynamic programming. In *Proceedings of the international joint conference on neural networks* (Vol. 2, pp. 1008–1013). doi:10.1109/IJCNN.2001.939498

Shervais, S., Shannon, T., & Lendaris, G. (2003). Intelligent supply chain management using adaptive critic learning. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, *33*(2), 235–244. doi:10.1109/TSMCA.2003.809214

Shin, J., & Lee, J. H. (2019). Multi-timescale, multi-period decision-making model development by combining reinforcement learning and mathematical programming. *Computers and Chemical Engineering*, *121*, 556–573. doi:10.1016/j.compchemeng.2018.11.020

Silver, E. A., Pyke, D. F., & Thomas, D. J. (2016). *Inventory and Production Management in Supply Chains* (4th editio). doi:10.1201/9781315374406

Snyder, L. V. (2018). *Multi-echelon base-stock optimization with upstream stockout costs*. Technical report, Lehigh University.

Sui, Z., Gosavi, A., & Lin, L. (2010). A reinforcement learning approach for inventory replenishment in vendor-managed inventory systems with consignment inventory. *EMJ - Engineering Management Journal*, *22*(4), 44–53. doi:10.1080/10429247.2010.11431878

Sutton, R. S. (1984). *Temporal Credit Assignment in Reinforcement Learning* (Doctoral dissertation, University of Massachusetts, Amherst).

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learing - An Introduction*.

Szepesvári, C. (2010). Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, *9*, 1–89. doi:10.2200/S00268ED1V01Y201005AIM009

Topan, E., Eruguz, A. S., Ma, W., Van Der Heijden, M. C., & Dekker, R. (2020). A review of operational spare parts service logistics in service control towers. *European Journal of Operational Research*, *282*(2), 401–414. doi:10.1016/j.ejor.2019.03.026

Van Der Heijden, M. C., Diks, E. B., & De Kok, A. G. (1997). Stock allocation in general multi-echelon distribution systems with (R, S) order-up-to-policies. *International Journal of Production Economics*, *49*(2), 157–174. doi:10.1016/S0925-5273(97)00005-4

Van Roy, B., Bertsekas, D. P., Lee, Y., & Tsitsiklis, J. N. (1997). A neuro-dynamic programming approach to retailer inventory management. *Proceedings of the IEEE Conference on Decision and Control*, *4*(December), 4052–4057. doi:10.1109/cdc.1997.652501

Van Santen, N. (2019). *Efficiently Engineering Variant Simulation Models to Enable Performance Evaluation of Different Inventory Systems*.

Van Tongeren, T., Kaymak, U., Naso, D., & Van Asperen, E. (2007). Q-learning in a competitive supply chain. *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, 1211–1216. doi:10.1109/ICSMC.2007.4414132

Vercraene, S., & Gayon, J. P. (2013). Optimal control of a production-inventory system with productreturns. *International Journal of Production Economics*, *142*(2), 302–310. doi:10.1016/j.ijpe.2012.11.012

Vermorel, J. (2015). Fill rate definition. Retrieved from https://www.lokad.com/fill-rate-definition

Violante, A. (2019). Simple Reinforcement Learning: Q-learning. Retrieved from https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards* (Doctoral dissertation, King's College).

Woerner, S., Laumanns, M., Zenklusen, R., & Fertis, A. (2015). Approximate dynamic programming for stochastic linear control problems on compact state spaces. *European Journal of Operational Research*, *241*(1), 85–98. doi:10.1016/j.ejor.2014.08.003

Xu, J., Zhang, J., & Liu, Y. (2009). An adaptive inventory control for a supply chain. *2009 Chinese Control and Decision Conference, CCDC 2009*, 5714–5719. doi:10.1109/CCDC.2009.5195218

Yang, S., & Zhang, J. (2015). Adaptive inventory control and bullwhip effect analysis for supply chains with non-stationary demand. *Proceedings of the 2015 27th Chinese Control and Decision Conference, CCDC 2015*, 3903–3908. doi:10.1109/CCDC.2015.7162605

Zarandi, M. H. F., Moosavi, S. V., & Zarinbal, M. (2013). A fuzzy reinforcement learning algorithm for inventory control in supply chains. *International Journal of Advanced Manufacturing Technology*, *65*(1-4), 557–569. doi:10.1007/s00170-012-4195-z

Zhang, K., Xu, J., & Zhang, J. (2013). A new adaptive inventory control method for supply chains with non-stationary demand. *2013 25th Chinese Control and Decision Conference, CCDC 2013*, 1034–1038. doi:10.1109/CCDC.2013.6561076

Zychlinski, S. (2019). Qrash Course: Reinforcement Learning 101 & Deep Q Networks in 10 Minutes. Retrieved from https://towardsdatascience.com/qrash-course-deep-q-networks-from-the-ground-up-1bbda41d3677

The icons that are used in Figure 1.1, Figure 2.3, Figure 3.2, Figure 4.3 and Figure 4.4 are adapted versions from:

- *Customer* by **Priyanka** from The Noun Project
- *Warehouse* by **RomanP** from The Noun Project
- *Truck* by **Anna Sophie** from The Noun Project
- *Factory* by **Iconsphere** from The Noun Project

A. Simulation Environment

```python
import random
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import gym
from gym import spaces


def generate_leadtime(t, dist, lowerbound, upperbound):
    """
    Generate the leadtime of the dataset from paper or distribution.

    Returns: Integer
    """
    if dist == 'uniform':
        leadtime = random.randrange(lowerbound, upperbound + 1)
    else:
        raise Exception
    return leadtime

class InventoryEnv(gym.Env):
    """
    General Inventory Control Environment.

    Currently tested with:
    - A reinforcement learning model for supply chain ordering management:
      An application to the beer game - Chaharsooghi (2002)
    """

    def __init__(self, case, action_low, action_high, action_min, action_max,
                 state_low, state_high, method,
                 coded=False, fix=True, ipfix=True):
        self.case = case
        self.case_name = case.__class__.__name__
        self.n = case.leadtime_ub + 1
        self.coded = coded
        self.fix = fix
        self.ipfix = ipfix
        self.method = method
        if self.method == 'DRL':
            self.action_low = action_low
            self.action_high = action_high
            self.action_min = action_min
            self.action_max = action_max
            self.state_low = state_low
            self.state_high = state_high
            self.determine_potential_actions()
            self.determine_potential_states()

    def determine_potential_actions(self):
        """
        Possible actions returned as Gym Space
        each period
        """
        self.feasible_actions = 0
        self.action_space = spaces.Box(self.action_low, self.action_high, dtype=np.int32)

    def determine_potential_states(self):
```

```
        """
        Based on the mean demand, we determine the maximum and minimum
        inventory to prevent the environment from reaching unlikely states
        """
        # Observation space consists of the current timestep and inventory positions of every echelon
        self.observation_space = spaces.Box(self.state_low, self.state_high, dtype=np.int32)

    def _generate_demand(self):
        """
        Generate the demand using a predefined distribution.

        Writes the demand to the orders table.
        """
        source, destination = np.nonzero(self.case.connections)
        for retailer, customer in zip(source[-self.case.no_customers:],
                                      destination[-self.case.no_customers:]):
            if self.case.demand_dist == 'poisson':
                demand_mean = random.randrange(self.case.demand_lb,
                                               self.case.demand_ub + 1)
                demand = np.random.poisson(demand_mean)
            elif self.case.demand_dist == 'uniform':
                demand = random.randrange(self.case.demand_lb,
                                          self.case.demand_ub + 1)
            self.O[0, customer, retailer] = demand

    def calculate_reward(self):
        """
        Calculate the reward for the current period.

        Returns: holding costs, backorder costs
        """
        backorder_costs = np.sum(self.BO[0] * self.case.bo_costs)
        holding_costs = np.sum(self.INV[0] * self.case.holding_costs)
        return holding_costs, backorder_costs

    def _initialize_state(self):
        """
        Initialize the inventory position for every node.

        Copies the inventory position from the previous timestep.
        """
        (...)

    def _receive_incoming_delivery(self):
        """
        Receives the incoming delivery for every stockpoint.

        Customers are not taken into account because of zero lead time
        Based on the amount stated in T
        """
        # Loop over all suppliers and stockpoints
        for i in range(0, self.case.no_stockpoints + self.case.no_suppliers):
            # Loop over all stockpoints
            # Note that only forward delivery is possible, hence 'i+1'
            for j in range(i + 1, self.case.no_stockpoints +
                           self.case.no_suppliers):
                delivery = self.T[0, i, j]
                self.INV[0, j] += delivery
                self.in_transit[0, i, j] -= delivery
                self.T[0, i, j] = 0

```

```
    def _receive_incoming_orders(self):
        # Loop over every stockpoint
        for i in range(self.case.no_stockpoints + self.case.no_suppliers):
            # Check if the inventory is larger than all incoming orders
            if self.INV[0, i] >= np.sum(self.O[0, :, i], 0):
                for j in np.nonzero(self.case.connections[i])[0]:
                    if self.O[0, j, i] > 0:
                        self._fulfill_order(i, j, self.O[0, j, i])
                        if self.t >= self.case.warmup:
                            self.TotalFulfilled[j,i] += self.O[0,j,i]
            else:
                IPlist = {}
                # Generate a list of stockpoints that have outstanding orders
                k_list = np.nonzero(self.O[0, :, i])[0]
                bo_echelon = np.sum(self.BO[0], 0)
                for k in k_list:
                    IPlist[k] = self.INV[0, k] - bo_echelon[k]
                # Check the lowest inventory position and sort these on lowest IP
                sorted_IP = {k: v for k, v in sorted(IPlist.items(), key=lambda item: item[1])}
                for j in sorted_IP:
                    inventory = self.INV[0, i]
                    # Check if the remaining order can be fulfilled completely
                    if inventory >= self.O[0, j, i]:
                        self._fulfill_order(i, j, self.O[0, j, i])
                        if self.t >= self.case.warmup:
                            self.TotalFulfilled[j,i] += self.O[0,j,i]
                    else:
                    # Else, fulfill how far possible
                        quantity = self.O[0, j, i] - inventory
                        self._fulfill_order(i, j, inventory)
                        if self.t >= self.case.warmup:
                            self.TotalFulfilled[j,i] += inventory
                        if self.case.unsatisfied_demand == 'backorders':
                            self.BO[0, j, i] += quantity
                            if self.t >= self.case.warmup:
                                self.TotalBO[j,i] += quantity
        if self.case.unsatisfied_demand == 'backorders':
            i_list, j_list = np.nonzero(self.case.connections)
            for i, j in zip(i_list, j_list):
                inventory = self.INV[0, i]
                # If there are any backorders, fulfill them afterwards
                if inventory > 0:
                    # If the inventory is larger than the backorder
                    # Fulfill the whole backorder
                    backorder = self.BO[0, j, i]
                    if inventory >= backorder:
                        if self.fix:
                            self._fulfill_order(i, j, backorder)
                        else:
                            self.INV[0, i] -= backorder
                        self.BO[0, j, i] = 0
                    # Else, fulfill the entire inventory
                    else:
                        self._fulfill_order(i, j, inventory)
                        self.BO[0, j, i] -= inventory

    def _recieve_incoming_orders_customers(self):
        i_list, j_list = np.nonzero(self.case.connections)
        for i, j in zip(i_list[-self.case.no_customers:], j_list[-self.case.no_customers:]):
            if self.O[0, j, i] > 0:
                # Check if the current order can be fulfilled
```

```
                if self.INV[0, i] >= self.O[0, j, i]:
                    self._fulfill_order(i, j, self.O[0, j, i])
                    # Else, fulfill as far as possible
                else:
                    inventory = max(self.INV[0, i], 0)
                    quantity = self.O[0, j, i] - inventory
                    self._fulfill_order(i, j, inventory)
                    # Add to backorder if applicable
                    if self.case.unsatisfied_demand == 'backorders':
                        self.BO[0, j, i] += quantity
        if self.case.unsatisfied_demand == 'backorders':
            for i, j in zip(i_list[-self.case.no_customers:], j_list[-self.case.no_customers:]):
                inventory = self.INV[0, i]
                # If there are any backorders, fulfill them afterwards
                if inventory > 0:
                    # If the inventory is larger than the backorder
                    # Fulfill the whole backorder
                    backorder = self.BO[0, j, i]
                    if inventory >= backorder:
                        # Dit vind ik heel onlogisch, maar voorzover ik nu kan zien
                        # in de IPs komt de backorder nooit aan.
                        # Nu wel gedaan dmv fix
                        if self.fix:
                            self._fulfill_order(i, j, backorder)
                        else:
                            self.INV[0, i] -= backorder
                        self.BO[0, j, i] = 0
                    # Else, fulfill the entire inventory
                    else:
                        self._fulfill_order(i, j, inventory)
                        self.BO[0, j, i] -= inventory

    def _recieve_incoming_orders_divergent(self):
        # Ship from supplier to warehouse
        self._fulfill_order(0, 1, self.O[0, 1, 0])
        # Check if the warehouse can ship all orders
        if self.INV[0, 1] >= np.sum(self.O[0, :, 1], 0):
            i_list, j_list = np.nonzero(self.case.connections)
            for i, j in zip(i_list[self.case.no_suppliers:self.case.no_suppliers+
                                   self.case.no_stockpoints],
                            j_list[self.case.no_suppliers:self.case.no_suppliers+
                                   self.case.no_stockpoints]):
                if self.O[0, j, i] > 0:
                    self._fulfill_order(i, j, self.O[0, j, i])
        else:
            IPlist = {}
            i_list, _ = np.nonzero(self.O[0])
            bo_echelon = np.sum(self.BO[0], 0)
            for i in i_list:
                IPlist[i] = self.INV[0, i] - bo_echelon[i]
            # Check the lowest inventory position and sort these on lowest IP
            sorted_IP = {k: v for k, v in sorted(IPlist.items(), key=lambda item: item[1])}
            # Check if there is still inventory left
            if self.INV[0, 1] >= 0:
                for i in sorted_IP:
                    # Check if the remaining order can be fulfilled completely
                    if self.INV[0, 1] >= self.O[0, i, 1]:
                        self._fulfill_order(1, i, self.O[0, i, 1])
                    else:
                    # Else, fulfill how far possible
                        inventory = max(self.INV[0, 1], 0)
```

```
                        quantity = self.O[0, i, 1] - inventory
                        self._fulfill_order(1, i, inventory)
                        break

    def _fulfill_order(self, source, destination, quantity):
        # Customers don't have any lead time.
        if destination >= self.case.no_nodes - self.case.no_customers:
            leadtime = 0
        else:
            leadtime = self.leadtime
        # The order is fulfilled immediately for the customer
        # or whenever the leadtime is 0
        if leadtime == 0:
            # The new inventorylevel is increased with the shipped quantity
            self.INV[0, destination] += quantity
        else:
            # If the order is not fulfilled immediately, denote the time when
            # the order will be delivered. This can not be larger than the horizon
            if leadtime < self.n:
                self.T[leadtime, source, destination] += quantity
            else:
                raise NotImplementedError
            for k in range(0, min(leadtime, self.n) + 1):
                self.in_transit[k, source, destination] += quantity
        # Suppliers have unlimited capacity
        if source >= self.case.no_suppliers:
            self.INV[0, source] -= quantity

    def _place_outgoing_order(self, t, action):
        k = 0
        incomingOrders = np.sum(self.O[0], 0)
        # Loop over all suppliers and stockpoints
        for j in range(self.case.no_suppliers, self.case.no_stockpoints +
                       self.case.no_suppliers):
            RandomNumber = random.random()
            probability = 0
            for i in range(0, self.case.no_stockpoints + self.case.no_suppliers):
                if self.case.connections[i, j] == 1:
                    self._place_order(i,j,t,k, action, incomingOrders)
                    k += 1
                elif self.case.connections[i,j] > 0:
                    probability += self.case.connections[i,j]
                    if RandomNumber < probability:
                        self._place_order(i,j,t,k, action, incomingOrders)
                        k += 1
                        break

    def _place_order(self, i, j, t, k, action, incomingOrders):
        if self.case.order_policy == 'X':
            self.O[t, j, i] += action[k]
            if (self.t < self.case.horizon - 1) and (self.t >= self.case.warmup-1):
                self.TotalDemand[j,i] += action[k]
        elif self.case.order_policy == 'X+Y':
            self.O[t, j, i] += incomingOrders[j] + action[k]
            if (self.t < self.case.horizon - 1) and (self.t >= self.case.warmup-1):
                self.TotalDemand[j,i] += incomingOrders[j] + action[k]
        elif self.case.order_policy == 'BaseStock':
            bo_echelon = np.sum(self.BO[0], 0)
            self.O[t, j, i] += max(0, action[k]-(self.INV[0,j]+self.in_transit[0,i,j]-bo_echelon[j]))
            if (self.t < self.case.horizon - 1) and (self.t >= self.case.warmup-1):
                self.TotalDemand[j,i] += max(0,action[k]-self.INV[0,j]+self.in_transit[0,i,j]-bo_echelon[j])
```

```
303         else:
304             raise NotImplementedError
305 
306     def _code_state(self):
307         (...)
308         return CIP
309 
310     def _check_action_space(self, action):
311         if isinstance(self.action_space, spaces.Box):
312             low = self.action_space.low
313             high = self.action_space.high
314             max = self.action_max
315             min = self.action_min
316             action_clip = np.clip(action, low, high)
317             for i in range(len(action_clip)):
318                 action_clip[i] = ((action_clip[i]-low[i])/(high[i]-low[i]))*((max[i]-min[i]))+min[i]
319             action = [np.round(num) for num in action_clip]
320         return action
321 
322     def step(self, action, visualize=False):
323         """
324         Execute one step in the RL method.
325 
326         input: actionlist, visualize
327         """
328         self.leadtime = generate_leadtime(0, self.case.leadtime_dist,
329                                           self.case.leadtime_lb, self.case.leadtime_ub)
330         action, penalty = self._check_action_space(action)
331         self._initialize_state()
332         if visualize: self._visualize("0. IP")
333         if self.case_name == "BeerGame" or self.case_name == "General":
334             self._generate_demand()
335             self._receive_incoming_delivery()
336             if visualize: self._visualize("1. Delivery")
337             self._receive_incoming_orders()
338             if visualize: self._visualize("2. Demand")
339             self._place_outgoing_order(1, action)
340         elif self.case_name == "Divergent":
341             # According to the paper:
342             # (1) Warehouse places order to external supplier
343             self._place_outgoing_order(0, action)
344             if visualize: self._visualize("1. Warehouse order")
345             # (2) Warehouse ships the orders to retailers taking the inventory position into account
346             self._recieve_incoming_orders_divergent()
347             if visualize: self._visualize("2. Warehouse ships")
348             # (3) Warehouse and retailers receive their orders
349             self._receive_incoming_delivery()
350             if visualize: self._visualize("3. Orders received")
351             # (4) Demand from customers is observed
352             self._generate_demand()
353             self._recieve_incoming_orders_customers()
354             if visualize: self._visualize("4. Demand")
355         else:
356             raise NotImplementedError
357         CIP = self._code_state()
358         holding_costs, backorder_costs = self.calculate_reward()
359         reward = holding_costs + backorder_costs + penalty
360         return CIP, -reward/self.case.divide, False
361 
362     def reset(self):
363         (...)
```

B. Beer Game - Q-learning Code

```
"""@author: KevinG."""
import random
import time as ct
import numpy as np
from inventory_env import InventoryEnv
from cases.beergame import BeerGame


def encode_state(state):
    """Encode the state, so we can find it in the q_table."""
    encoded_state = (state[0] - 1) * 729
    encoded_state += (state[1] - 1) * 81
    encoded_state += (state[2] - 1) * 9
    encoded_state += (state[3] - 1)
    return int(encoded_state)

def decode_action(action):
    """Decode the action, so we can use it in the environment."""
    decoded_action = []
    decoded_action.append(int(action / 64))
    action = action % 64
    decoded_action.append(int(action / 16))
    action = action % 16
    decoded_action.append(int(action / 4))
    action = action % 4
    action = decoded_action.append(int(action))
    return decoded_action

def get_next_action_paper(time):
    """
    Determine the next action to take based on the policy from the paper.
    """
    actionlist = [106, 247, 35, 129, 33, 22, 148, 22, 231, 22, 22, 111, 111,
                  229, 192, 48, 87, 49, 0, 187, 254, 236, 94, 94, 89, 25, 22,
                  250, 45, 148, 106, 243, 151, 123, 67]
    return actionlist[time]

class QLearning:
    """Based on the beer game by Chaharsooghi (2008)."""

    def __init__(self, seed):
        random.seed(seed)
        np.random.seed(seed)
        self.case                           = BeerGame()
        self.dist                           = 'uniform'
        # State-Action variables
        possible_states = [1, 2, 3, 4, 5, 6, 7, 8, 9]
        possible_actions = [0, 1, 2, 3]

        self.no_states = len(possible_states) ** self.case.no_stockpoints
        self.no_actions = len(possible_actions) ** self.case.no_stockpoints

        self.initialize_q_learning()

        # Initialize environment
        self.env = InventoryEnv(case=self.case, n=self.horizon, action_low=0, action_high=0,
                                action_min=0, action_max=0, state_low=0, state_high=0,
                                actions=self.no_actions, coded=True, fix=False, ipfix=False,
```

```
                                method='Q-learning')

    def initialize_q_learning(self):
        # Time variables
        self.max_iteration      = 1000000     # max number of iterations
        self.alpha              = 0.17        # Learning rate
        self.horizon            = 35
        self.stepsize           = 1000

        # Exploration Variables
        self.exploitation       = 0.02        # Starting exploitation rate
        exploitation_max        = 0.90
        exploitation_min        = 0.02
        self.exploitation_iter_max = 0.98
        self.exploitation_delta = ((exploitation_max -
                                    exploitation_min) /
                                   (self.max_iteration - 1))
        # Initialize Q values
        self.q_table = np.full([self.horizon + 1, self.no_actions, self.no_states], -10000)


    def get_next_action(self, time, state, exploitation):
        """Determine the next action to be taken.

        Based on the exploitation rate
        Returns a list of actions
        """
        if random.random() <= exploitation:
            return self.greedy_action(time, state)
        return self.random_action()

    def greedy_action(self, time, state):
        """Retrieve the best action for the current state.

        Picks the best action corresponding to the highest Q value
        Returns a list of actions
        """
        state_e = encode_state(state)
        action = self.q_table[time][:, state_e].argmax()
        return action

    def random_action(self):
        """Generate a random set of actions."""
        action = random.randint(0, self.no_actions - 1)
        return action

    def get_q(self, time, state_e):
        """Retrieve highest Q value of the state in the next time period."""
        return self.q_table[time + 1][:, state_e].max()

    def update(self, time, old_state, new_state, action, reward):
        """Update the Q table."""
        new_state_e = encode_state(new_state)
        old_state_e = encode_state(old_state)

        new_state_q_value = self.get_q(time, new_state_e)
        old_state_q_value = self.q_table[time][action, old_state_e]
        if new_state_q_value == -10000: new_state_q_value = 0
        if old_state_q_value == -10000: old_state_q_value = 0
        q_value = self.alpha * (reward + new_state_q_value - old_state_q_value)
        # q_value = self.alpha * (reward + new_state_q_value - self.q_table[time][0, old_state_e])
```

```
120         if self.q_table[time][action, old_state_e] == -10000:
121             self.q_table[time][action, old_state_e] = q_value
122         else:
123             self.q_table[time][action, old_state_e] += q_value
124         # self.q_table[time][0, old_state_e] += q_value
125 
126     def iteration(self):
127         """Iterate over the simulation."""
128         current_iteration, time = 0, 0
129         while current_iteration < self.max_iteration:
130             exploitation_iter = self.exploitation
131             exploitation_iter_delta = ((self.exploitation_iter_max -
132                                         self.exploitation) / (self.horizon - 1))
133             self.case.leadtime_dist, self.case.demand_dist = self.dist, self.dist
134             old_state = self.env.reset()
135             while time < self.horizon:
136                 # action = get_next_action_paper(time)
137                 action = self.get_next_action(time, old_state, exploitation_iter)
138                 # Take action and calculate r(t+1)
139                 action_d = decode_action(action)
140                 new_state, reward, _, _ = self.env.simulate(action_d)
141                 self.update(time, old_state, new_state, action, reward)
142                 old_state = new_state
143                 exploitation_iter += exploitation_iter_delta
144                 time += 1
145             self.exploitation += self.exploitation_delta
146             time = 0
147             current_iteration += 1
148 
149 
150 STARTTIME = ct.time()
151 for k in range(0, 10):
152     ENV = QLearning(k)
153     print("Replication " + str(k))
154     QLearning.iteration(ENV)
```

C. Beer Game - Q-learning Experiment

To limit the action space, we only define one action: $Y = 0$. This way, we make sure the Q-table has a small size, which makes it easier for the Q-learning algorithm to learn the correct values.

In Figure C.1, we can see the highest Q-value of the first state. The orange line represents the total costs of the simulation.

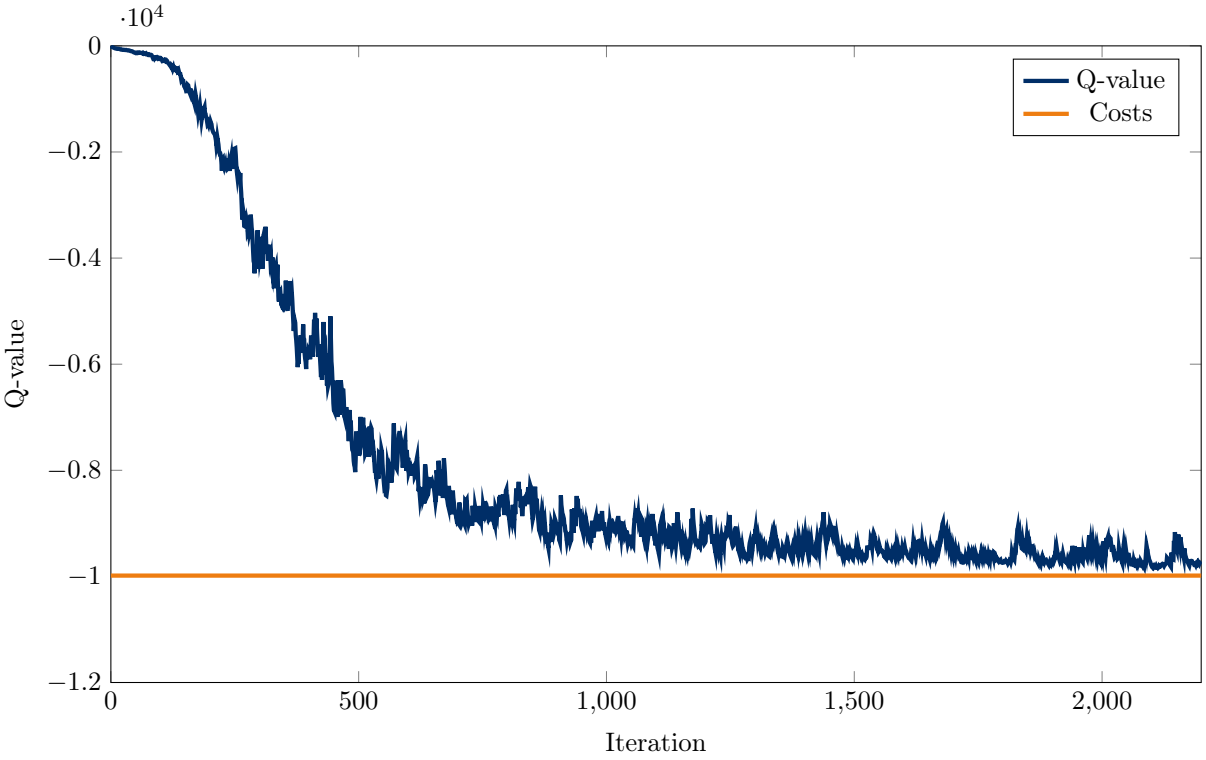


Figure C.1: *Highest Q-value of the first state per iteration.*

D. Beer Game - Q-Values

-4282.71	-4281.67	-4279.46	-4280.64	-4279.64	-4280.29	-4280.55	-4280.94
-4281.56	-4278.61	-4281.38	-4281.00	-4283.40	-4281.52	-4281.36	-4280.66
-4282.02	-4281.86	-4281.83	-4282.93	-4281.25	-4281.17	-4280.10	-4279.46
-4282.77	-4281.47	-4279.39	-4281.58	-4280.45	-4281.51	-4280.20	-4278.97
-4278.44	-4280.07	-4280.23	-4280.40	-4279.73	-4281.79	-4279.79	-4280.95
-4281.98	-4280.59	-4278.78	-4280.62	-4279.29	-4278.73	-4281.00	-4281.12
-4276.68	-4280.70	-4279.99	-4278.45	-4281.33	-4282.68	-4282.33	-4281.80
-4280.58	-4279.46	-4281.37	-4280.77	-4281.82	-4281.46	-4281.08	-4279.16
-4282.41	-4282.17	-4277.82	-4283.78	-4280.64	-4281.88	-4282.81	-4283.84
-4277.89	-4278.43	-4282.82	-4281.03	-4277.88	-4280.13	-4280.30	-4277.74
-4279.19	-4279.30	-4280.98	-4278.93	-4278.18	-4279.71	-4281.70	-4279.93
-4283.53	-4280.73	-4275.43	-4281.32	-4282.57	-4281.48	-4279.19	-4280.78
-4281.40	-4280.60	-4278.24	-4280.72	-4281.89	-4279.74	-4280.14	-4282.57
-4283.38	-4278.64	-4281.19	-4279.18	-4280.96	-4281.41	-4279.31	-4282.19
-4279.31	-4280.14	-4281.58	-4279.56	-4275.69	-4279.18	-4276.17	-4279.69
-4282.56	-4281.12	-4279.76	-4279.44	-4280.21	-4282.06	-4279.65	-4281.71
-4279.52	-4281.18	-4279.76	-4280.95	-4277.45	-4277.43	-4282.13	-4279.39
-4278.08	-4280.56	-4278.25	-4281.57	-4279.77	-4280.03	-4281.66	-4280.40
-4282.13	-4281.42	-4283.61	-4281.19	-4279.39	-4278.88	-4281.01	-4280.67
-4279.09	-4283.33	-4281.72	-4283.73	-4281.03	-4281.04	-4280.24	-4279.56
-4278.02	-4281.01	-4279.47	-4280.69	-4281.69	-4278.85	-4280.57	-4280.38
-4284.04	-4282.18	-4281.98	-4281.03	-4284.11	-4281.41	-4276.85	-4279.35
-4282.74	-4280.43	-4280.23	-4276.90	-4278.11	-4278.50	-4280.50	-4279.83
-4281.84	-4277.83	-4281.85	-4281.56	-4282.90	-4278.30	-4278.82	-4284.42
-4282.31	-4282.99	-4282.00	-4280.70	-4281.42	-4282.18	-4283.85	-4282.25
-4274.26	-4279.44	-4281.89	-4282.19	-4280.16	-4279.75	-4283.44	-4285.21
-4277.25	-4280.70	-4279.13	-4281.25	-4282.52	-4281.52	-4279.17	-4283.03
-4279.43	-4282.92	-4282.23	-4279.29	-4274.61	-4282.25	-4282.39	-4282.72
-4281.19	-4281.89	-4282.44	-4278.73	-4279.50	-4280.04	-4280.34	-4279.75
-4282.32	-4281.60	-4279.52	-4279.60	-4278.30	-4282.09	-4281.86	-4281.81
-4277.59	-4281.87	-4279.09	-4282.18	-4283.99	-4278.41	-4278.31	-4280.36
-4280.87	-4279.14	-4282.76	-4280.33	-4278.76	-4280.36	-4285.25	-4281.72

Table D.1: *Q-values of the first state (10 replications).*

E. Beer Game - DRL Code

```
class BeerGame:
    """Based on the beer game by Chaharsooghi et al (2008)."""

    def __init__(self):
        # Supply chain variables
        # Number of nodes per echelon, including suppliers and customers
        # The first element is the number of suppliers
        # The last element is the number of customers
        self.stockpoints_echelon   = [1, 1, 1, 1, 1, 1]
        # Number of suppliers
        self.no_suppliers          = self.stockpoints_echelon[0]
        # Number of customers
        self.no_customers          = self.stockpoints_echelon[-1]
        # Number of stockpoints
        self.no_stockpoints = sum(self.stockpoints_echelon) - \
            self.no_suppliers - self.no_customers

        # Total number of nodes
        self.no_nodes              = sum(self.stockpoints_echelon)
        # Total number of echelons, including supplier and customer
        self.no_echelons           = len(self.stockpoints_echelon)

        # Connections between every stockpoint
        self.connections           = np.array([
                                       [0, 1, 0, 0, 0, 0],
                                       [0, 0, 1, 0, 0, 0],
                                       [0, 0, 0, 1, 0, 0],
                                       [0, 0, 0, 0, 1, 0],
                                       [0, 0, 0, 0, 0, 1],
                                       [0, 0, 0, 0, 0, 0]
                                      ])
        # Determines what happens with unsatisfied demand, can be either 'backorders' or 'lost_sales'
        self.unsatisfied_demand    = 'backorders'
        # Initial inventory per stockpoint
        self.initial_inventory     = [100000, 12, 12, 12, 12, 0]
        # Holding costs per stockpoint
        self.holding_costs         = [0, 1, 1, 1, 1, 0]
        # Backorder costs per stockpoint
        self.bo_costs              = [2, 2, 2, 2, 2, 2]
        # Demand distribution, can be either 'poisson' or 'uniform'
        self.demand_dist           = 'uniform'
        # Lower bound of the demand distribution
        self.demand_lb             = 0
        # Upper bound of the demand distribution
        self.demand_ub             = 15
        # Leadtime distribution, can only be 'uniform'
        self.leadtime_dist         = 'uniform'
        # Lower bound of the leadtime distribution
        self.leadtime_lb           = 0
        # Upper bound of the leadtime distribution
        self.leadtime_ub           = 4
        # Predetermined order policy, can be either 'X' or 'X+Y'
        self.order_policy          = 'X'
        self.horizon               = 35
        self.divide                = False
        self.warmup                = 1
        self.fix                   = False
        self.action_low            = np.array([-1, -1, -1, -1])
```

```
        self.action_high           = np.array([1, 1, 1, 1])
        self.action_min            = np.array([0,0,0,0])
        self.action_max            = np.array([30,30,30,30])
        self.state_low             = np.zeros([50])
        self.state_high            = np.array([4000,4000,
                                               1000,1000,1000,1000,
                                               1000,1000,1000,1000,
                                               30,30,30,30,
                                               150,150,150,150,
                                               150,150,150,150,
                                               150,150,150,150,
                                               150,150,150,150,
                                               30,30,30,30,
                                               30,30,30,30,
                                               30,30,30,30,
                                               30,30,30,30])

# PPO Settings
# activation function of network
network_activation          = 'tanh'
# size of network
network_size                = (64, 64)
# initial values of bias in network
network_bias_init           = 0.0
# method of weight initialization for network (uniform or normal)
network_weights_init        = 'uniform'
# number of iterations between evaluation
ppo_evaluation_steps        = 100
#number of consecutive evaluation iterations without improvement
ppo_evaluation_threshold    = 250
# maximum number of iterations in learning run
ppo_iterations              = 25000
# length of one episode in buffer
ppo_buffer_length           = 256
# discount factor used in GAE calculations
ppo_gamma                   = 0.99
# lambda rate used in GAE calculations
ppo_lambda                  = 0.95
# indicator of using a cooldown period in the buffer (boolean)
cooldown_buffer             = False
# clipping value used in policy loss calculations
ppo_epsilon                 = 0.2
# learning rate for policy network
pi_lr                       = 1e-4
# learning rate for value network
vf_lr                       = 1e-4
# after x iterations, save  model weights and histograms to tensorboard
ppo_save_freq               = 100
# nr of epochs (i.e. repetitions of the buffer) used in updating the model weights
ppo_epochs                  = 10
# batch size used to split the buffer for updating the model weights
ppo_batch_size              = 64
# number of simulation runs to compute benchmark and as stopping criterion
ppo_simulation_runs         = 1
# length of simulation to compute benchmark and as stopping criterion
ppo_simulation_length       = 35
# length of initial simulation that is discarded
ppo_warmup_period           = 0

policy_results_states       = [[0,12,12,12,12]]
```

```
120
121 for k in range(10):
122     print("Replication " + str(k))
123     # Initialize environment
124     env = InventoryEnv(case, action_low, action_high, action_min, action_max,
125                        state_low, state_high, FIX, 'DRL')
126     run_name = "RN{}".format(k)
127
128     # set random seed
129     set_seeds(env, k)
130
131     # call learning function
132     ppo_learning(env, False, experiment_name, run_name,
133                  network_activation, network_size, network_bias_init, network_weights_init,
134                  ppo_evaluation_steps, ppo_evaluation_threshold,
135                  ppo_iterations, ppo_buffer_length, ppo_gamma, ppo_lambda, cooldown_buffer,
136                  ppo_epsilon, pi_lr, vf_lr, ppo_save_freq, ppo_epochs, ppo_batch_size,
137                  ppo_simulation_runs, ppo_simulation_length, ppo_warmup_period, policy_results_states)
```

F. Divergent - DRL Code

```
class Divergent:
    """ Based on the paper of Kunnumkal and Topaloglu (2011)"""

    def __init__(self):
        # Supply chain variables
        # Number of nodes per echelon, including suppliers and customers
        # The first element is the number of suppliers
        # The last element is the number of customers
        self.stockpoints_echelon            = [1, 1, 3, 3]
        # Number of suppliers
        self.no_suppliers = self.stockpoints_echelon[0]
        # Number of customers
        self.no_customers = self.stockpoints_echelon[-1]
        # Number of stockpoints
        self.no_stockpoints = sum(self.stockpoints_echelon) - \
            self.no_suppliers - self.no_customers

        # Total number of nodes
        self.no_nodes = sum(self.stockpoints_echelon)
        # Total number of echelons, including supplier and customer
        self.no_echelons = len(self.stockpoints_echelon)

        # Connections between every stockpoint
        self.connections                = np.array([
                                            [0, 1, 0, 0, 0, 0, 0, 0],
                                            [0, 0, 1, 1, 1, 0, 0, 0],
                                            [0, 0, 0, 0, 0, 1, 0, 0],
                                            [0, 0, 0, 0, 0, 0, 1, 0],
                                            [0, 0, 0, 0, 0, 0, 0, 1],
                                            [0, 0, 0, 0, 0, 0, 0, 0],
                                            [0, 0, 0, 0, 0, 0, 0, 0],
                                            [0, 0, 0, 0, 0, 0, 0, 0]
                                            ])
        # Determines what happens with unsatisfied demand, can be either 'backorders' or 'lost_sales'
        self.unsatisfied_demand             = 'backorders'
        # Initial inventory per stockpoint
        self.initial_inventory              = [1000000, 0, 0, 0, 0, 0, 0, 0]
        # Holding costs per stockpoint
        self.holding_costs                  = [0, 0.6, 1, 1, 1, 0, 0, 0]
        # Backorder costs per stockpoint
        self.bo_costs                       = [0, 0, 19, 19, 19, 0, 0, 0]
        # Demand distribution, can be either 'poisson' or 'uniform'
        self.demand_dist                    = 'poisson'
        # Lower bound of the demand distribution
        self.demand_lb                      = 5
        # Upper bound of the demand distribution
        self.demand_ub                      = 15
        # Leadtime distribution, can only be 'uniform'
        self.leadtime_dist                  = 'uniform'
        # Lower bound of the leadtime distribution
        self.leadtime_lb                    = 1
        # Upper bound of the leadtime distribution
        self.leadtime_ub                    = 1
        # Predetermined order policy, can be either 'X','X+Y' or 'BaseStock'
        self.order_policy                   = 'X'
        self.action_low                     = np.array([-1,-1,-1,-1])
        self.action_high                    = np.array([1,1,1,1])
        self.action_min                     = np.array([0,0,0,0])
```

```
        self.action_max                     = np.array([300,75,75,75])
        self.state_low                      = np.zeros([13])
        self.state_high                     = np.array([1000, 450,        # Total inventory and backorders
                                                        250,250,250,250,  # Inventory per stockpoint
                                                        150,150,150,      # Backorders per stockpoint
                                                        150,150,150,150]) # In transit per stockpoint
        self.horizon                        = 75
        self.warmup                         = 25
        self.divide                         = 1000

# PPO Settings
# activation function of network
network_activation          = 'tanh'
# size of network
network_size                = (64, 64)
# initial values of bias in network
network_bias_init           = 0.0
# method of weight initialization for network (uniform or normal)
network_weights_init        = 'uniform'
# number of iterations between evaluation
ppo_evaluation_steps        = 100
#number of consecutive evaluation iterations without improvement
ppo_evaluation_threshold    = 250
# maximum number of iterations in learning run
ppo_iterations              = 30000
# length of one episode in buffer
ppo_buffer_length           = 256
# discount factor used in GAE calculations
ppo_gamma                   = 0.99
# lambda rate used in GAE calculations
ppo_lambda                  = 0.95
# indicator of using a cooldown period in the buffer (boolean)
cooldown_buffer             = False
# clipping value used in policy loss calculations
ppo_epsilon                 = 0.2
# learning rate for policy network
pi_lr                       = 1e-4
# learning rate for value network
vf_lr                       = 1e-4
# after x iterations, save  model weights and histograms to tensorboard
ppo_save_freq               = 500
# nr of epochs (i.e. repetitions of the buffer) used in updating the model weights
ppo_epochs                  = 10
# batch size used to split the buffer for updating the model weights
ppo_batch_size              = 64
# number of simulation runs to compute benchmark and as stopping criterion
ppo_simulation_runs         = 100
# length of simulation to compute benchmark and as stopping criterion
ppo_simulation_length       = 75
# length of initial simulation that is discarded
ppo_warmup_period           = 25

policy_results_states           = [[0,12,12,12,12]]

for k in range(10):
    print("Replication " + str(k))
    # Initialize environment
    env = InventoryEnv(case, case.action_low, case.action_high,
                       case.action_min, case.action_max, case.state_low, case.state_high,
                       'DRL', fix=True)
    run_name = "RN{}".format(k)
```

```
120
121     # set random seed
122     set_seeds(env, k)
123
124     # call learning function
125     ppo_learning(env, False, experiment_name, run_name,
126                  network_activation, network_size, network_bias_init, network_weights_init,
127                  ppo_evaluation_steps, ppo_evaluation_threshold,
128                  ppo_iterations, ppo_buffer_length, ppo_gamma, ppo_lambda, cooldown_buffer,
129                  ppo_epsilon, pi_lr, vf_lr, ppo_save_freq, ppo_epochs, ppo_batch_size,
130                  ppo_simulation_runs, ppo_simulation_length, ppo_warmup_period, policy_results_states)
```

G. Divergent - Pseudo-code

Algorithm 8: Event 1. Previous orders are received from the upstream stock point

```
for i=0 to all stock points do
    for j=0 to all stock points and supplier do
        add the shipment of current time step t from j to i to the inventory of i
        remove the shipment from in_transit table
```

Algorithm 9: Event 2. Orders are received from downstream stock point

```
generate the random demand for customer from the Poisson distribution with λ = [5, 15]
for i=0 to warehouse and stock points do
    for j=0 to stock points and customers do
        if inventory of j ≥ order from i to j then
            fulfill the complete order                          /* See Algorithm 6 */
        else
            fulfill the on-hand inventory if possible           /* See Algorithm 6 */
            add remaining part of the order to backlog
for i=0 to warehouse and stock points do
    for j=0 to stock points and customers do
        if there are backorders for j and inventory of j > 0 then
            if inventory is larger than backorders then
                fulfill the entire inventory                    /* See Algorithm 6 */
                empty the backlog
            else
                fulfill on-hand inventory                       /* See Algorithm 6 */
                remove fulfilled amount from backlog
```

H. Divergent - Heuristic

In this appendix, we elaborate on the calculations for the heuristic of Rong et al. (2017) to determine the base-stock levels of the divergent inventory system. The formulas of Rong et al. (2017) are extended with our own calculations.

Under the DA heuristic, we decompose the OWMR system into N two-location serial systems, solve the base-stock levels of the locations in each serial system and aggregate the solutions utilizing a procedure we call "backorder matching." We use s_i^a to denote thelocal base-stock level at location i based on the DA heuristic. Next, we provide a detailed description of the steps of the heuristic.

Step 1

Decompose the system into N serial systems. Serial system i consists of the warehouse and retailer i. We use 0_i to refer to the warehouse in serial system i. Utilizing the procedure in Shang and Song (2003), we approximate the echelon base-stock levels of retailer i, S_i^{SS}, and the warehouse in serial system i, $S_0^{SS}i$, as follows:

$$\begin{aligned} S_i^{SS} &= F_{D_i}^{-1}\left(\frac{b_i+h_0}{b_i+h_i}\right) = F_{D_i}^{-1}(0.98) = 30 \\ S_{0_i}^{SS} &= \frac{1}{2}\left[G_{\tilde{D}_i}^{-1}\left(\frac{b_i}{b_i+h_i}\right) + G_{\tilde{D}_i}^{-1}\left(\frac{b_i}{b_i+h_0}\right)\right]\frac{1}{2}\left[G_{\tilde{D}_i}^{-1}(0.95) + G_{\tilde{D}_i}^{-1}(0.97)\right] = 52 \end{aligned} \tag{H.1}$$

Here, $\tilde{D}_i$ is the total leadtime demand in serial system i. D_i is a Poisson random variable with rate $\lambda_i(L_0+L_i)$. F^{-1} is the inverse Poisson cumulative distribution function (CDF). For the warehouse, instead of using F^{-1}, we opt instead to use G_D^{-1}, the inverse function of an approximate Poisson CDF. Let $\lceil x \rceil$ and $\lfloor x \rfloor$ be the smallest integer no less than x and the largest integer no greater than x, respectively. We define this approximate CDF of a Poisson random variable D to be the following continuous, piecewise linear function:

$$G_D(x) = \begin{cases} 2F_D(0)x, & \text{if } x \leq 0.5 \\ F_D(\lfloor x-0.5 \rfloor) + [F_D(\lceil x-0.5 \rceil) & \\ -F_D(\lfloor x-0.5 \rfloor)](x-0.5 & \text{if } x \geq 0.5 \\ -\lfloor x-0.5 \rfloor) & \end{cases} \tag{H.2}$$

Step 2

Calculate the local base-stock level for the warehouse in serial system i by $s_{0_1}^d = S_{0_1}^{SS} - S_i^{SS}$. For the retailers, we have $s_i^d = S_i^{\zeta S}, \forall i \in \{1,2,\ldots,N\}$ We approximate the expected backorders of the warehouse in serial system i by

$$E[B_{0_i}] = E\left[\left(D_{0_i} - s_{0_i}^d\right)^+\right] = Q_{D_{0_i}}\left(s_{0_i}^d\right) = 0.97949 \tag{H.3}$$

where D_0 is a Poisson random variable with rate $\lambda_i L_0$, and $Q_D(x)$ is the loss function of the Poisson random variable D.

Step 3

Aggregate the serial systems back into the OWMR system utilizing a "backorder matching" procedure. We approximate the total expected backorders at the warehouse by $E[B_0] \cong \Sigma_{i=1}^N E[B_0]$. Specifically,

the backorder matching procedure sets s_0^a, the base-stock level at the warehouse, equal to the smallest integer s_0 such that:

$$E\left[(D_0 - s_0)^+\right] \leq \sum_{i=1}^{N} E\left[B_{0_i}\right] = 2.9384 \tag{H.4}$$

In other words,

$$s_0^a = Q_{D_0}^{-1}\left(\sum_{i=i}^{N} Q_{D_{0_i}}\left(s_{0_i}^d\right)\right) = 124 \tag{H.5}$$

where $Q_D^{-1}(y)$ is defined as $\min\{s \in \mathbb{Z} : Q_D(s) \leq y\}$ Similar to Theorem 1 for RO, for OWMR systems, we now prove the asymptotic optimality of DA as h_0 goes to 0.

This gives us the base-stock parameters: [124, 30, 30, 30].

I. CBC - DRL Code

```
class General:
    """ Based on the case of the CardBoard Company """

    def __init__(self):
        # Supply chain variables
        # Number of nodes per echelon, including suppliers and customers
        # The first element is the number of suppliers
        # The last element is the number of customers
        self.stockpoints_echelon            = [4, 4, 5, 5]
        # Number of suppliers
        self.no_suppliers = self.stockpoints_echelon[0]
        # Number of customers
        self.no_customers = self.stockpoints_echelon[-1]
        # Number of stockpoints
        self.no_stockpoints = sum(self.stockpoints_echelon) - \
            self.no_suppliers - self.no_customers

        # Total number of nodes
        self.no_nodes = sum(self.stockpoints_echelon)
        # Total number of echelons, including supplier and customer
        self.no_echelons = len(self.stockpoints_echelon)
        # Connections between every stockpoint
        self.connections = np.array([
                                #0  1  2  3  4  5  6  7    8    9    10   11   12 13 14 15 16 17
                                [0, 0, 0, 0, 1, 0, 0, 0,   0,   0,    0,   0,   0, 0, 0, 0, 0, 0],   # 0
                                [0, 0, 0, 0, 0, 1, 0, 0,   0,   0,    0,   0,   0, 0, 0, 0, 0, 0],   # 1
                                [0, 0, 0, 0, 0, 0, 1, 0,   0,   0,    0,   0,   0, 0, 0, 0, 0, 0],   # 2
                                [0, 0, 0, 0, 0, 0, 0, 1,   0,   0,    0,   0,   0, 0, 0, 0, 0, 0],   # 3
                                [0, 0, 0, 0, 0, 0, 0, 0, 0.6, 0.5, 0.15,   0,   0, 0, 0, 0, 0, 0],   # 4
                                [0, 0, 0, 0, 0, 0, 0, 0, 0.3, 0.4, 0.80, 0.1,   0, 0, 0, 0, 0, 0],   # 5
                                [0, 0, 0, 0, 0, 0, 0, 0,   0,   0,    0, 0.8, 0.7, 0, 0, 0, 0, 0],   # 6
                                [0, 0, 0, 0, 0, 0, 0, 0, 0.1, 0.1, 0.05, 0.1, 0.3, 0, 0, 0, 0, 0],   # 7
                                [0, 0, 0, 0, 0, 0, 0, 0,   0,   0,    0,   0,   0, 1, 0, 0, 0, 0],   # 8
                                [0, 0, 0, 0, 0, 0, 0, 0,   0,   0,    0,   0,   0, 0, 1, 0, 0, 0],   # 9
                                [0, 0, 0, 0, 0, 0, 0, 0,   0,   0,    0,   0,   0, 0, 0, 1, 0, 0],   # 10
                                [0, 0, 0, 0, 0, 0, 0, 0,   0,   0,    0,   0,   0, 0, 0, 0, 1, 0],   # 11
                                [0, 0, 0, 0, 0, 0, 0, 0,   0,   0,    0,   0,   0, 0, 0, 0, 0, 1],   # 12
                                [0, 0, 0, 0, 0, 0, 0, 0,   0,   0,    0,   0,   0, 0, 0, 0, 0, 0],   # 13
                                [0, 0, 0, 0, 0, 0, 0, 0,   0,   0,    0,   0,   0, 0, 0, 0, 0, 0],   # 14
                                [0, 0, 0, 0, 0, 0, 0, 0,   0,   0,    0,   0,   0, 0, 0, 0, 0, 0],   # 15
                                [0, 0, 0, 0, 0, 0, 0, 0,   0,   0,    0,   0,   0, 0, 0, 0, 0, 0],   # 16
                                [0, 0, 0, 0, 0, 0, 0, 0,   0,   0,    0,   0,   0, 0, 0, 0, 0, 0]    # 17
                                ])
        # Determines what happens with unsatisfied demand, can be either 'backorders' or 'lost_sales'
        self.unsatisfied_demand     = 'backorders'
        # Holding costs per stockpoint
        self.holding_costs          = [0, 0, 0, 0, 0.6, 0.6, 0.6, 0.6, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
        # Backorder costs per stockpoint
        self.bo_costs               = [0, 0, 0, 0, 0, 0, 0, 0, 19, 19, 19, 19, 19, 0, 0, 0, 0, 0]
        # Demand distribution, can be either 'poisson' or 'uniform'
        self.demand_dist            = 'poisson'
        # Lower bound of the demand distribution
        self.demand_lb              = 15
        # Upper bound of the demand distribution
        self.demand_ub              = 15
        # Leadtime distribution, can only be 'uniform'
        self.leadtime_dist          = 'uniform'
        # Lower bound of the leadtime distribution
```

```
        self.leadtime_lb            = 1
        # Upper bound of the leadtime distribution
        self.leadtime_ub            = 1
        # Predetermined order policy, can be either 'X' or 'X+Y' or 'BaseStock'
        self.order_policy           = 'X'
        self.horizon                = 75
        self.warmup                 = 50
        self.divide                 = 1000
        self.action_low             = np.array([-5,-5,-5,-5,-5,-5,-5,-5,-5])
        self.action_high            = np.array([5,5,5,5,5,5,5,5,5])
        self.action_min             = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0])
        self.action_max             = np.array([300, 300, 300, 300, 75, 75, 75, 75, 75])
        self.state_low              = np.zeros(48)
        self.state_high             = np.array([4500, 7750,     # Total inventory and backorders
                    500,500,500,500,500,500,500,500,500,     # Inventory per stockpoint
                    #11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
                    500,500,500,500,500,500,500,500,500,500,500,500,500,150,150,150,150,150,
                    #30, 31, 32, 33,34,35,36,37,38,39,40,41,42,43,44,45,46,47
                    150,150,150,150,75,75,75,75,75,75,75,75,75,75,75,75,75]) # In transit per stockpoint

# PPO Settings
# activation function of network
network_activation          = 'tanh'
# size of network
network_size                = (64, 64)
# initial values of bias in network
network_bias_init           = 0.0
# method of weight initialization for network (uniform or normal)
network_weights_init        = 'uniform'
# number of iterations between evaluation
ppo_evaluation_steps        = 500
#number of consecutive evaluation iterations without improvement
ppo_evaluation_threshold    = 250
# maximum number of iterations in learning run
ppo_iterations              = 50000
# length of one episode in buffer
ppo_buffer_length           = 256
# discount factor used in GAE calculations
ppo_gamma                   = 0.99
# lambda rate used in GAE calculations
ppo_lambda                  = 0.95
# indicator of using a cooldown period in the buffer (boolean)
cooldown_buffer             = False
# clipping value used in policy loss calculations
ppo_epsilon                 = 0.2
# learning rate for policy network
pi_lr                       = 1e-4
# learning rate for value network
vf_lr                       = 1e-4
# after x iterations, save  model weights and histograms to tensorboard
ppo_save_freq               = 500
# nr of epochs (i.e. repetitions of the buffer) used in updating the model weights
ppo_epochs                  = 10
# batch size used to split the buffer for updating the model weights
ppo_batch_size              = 64
# number of simulation runs to compute benchmark and as stopping criterion
ppo_simulation_runs         = 100
# length of simulation to compute benchmark and as stopping criterion
ppo_simulation_length       = case.horizon
# length of initial simulation that is discarded
ppo_warmup_period           = case.warmup
```

```
120
121 policy_results_states           = [[0,12,12,12,12]]
122
123 for k in range(10):
124     print("Replication " + str(k))
125     # Initialize environment
126     env = InventoryEnv(case, case.action_low, case.action_high,
127                        case.action_min, case.action_max, case.state_low, case.state_high,
128                        'DRL', fix=True)
129     run_name = "RN{}".format(k)
130
131     # set random seed
132     set_seeds(env, k)
133
134     # call learning function
135     ppo_learning(env, False, experiment_name, run_name,
136                  network_activation, network_size, network_bias_init, network_weights_init,
137                  ppo_evaluation_steps, ppo_evaluation_threshold,
138                  ppo_iterations, ppo_buffer_length, ppo_gamma, ppo_lambda, cooldown_buffer,
139                  ppo_epsilon, pi_lr, vf_lr, ppo_save_freq, ppo_epochs, ppo_batch_size,
140                  ppo_simulation_runs, ppo_simulation_length, ppo_warmup_period, policy_results_states)
```