Technical University of
Munich

Department of Informatics

Bachelor’s Thesis in Informatics

Reinforcement Learning in the MIT Beer Distribution Game

Author:	Daniel Schroter
Supervisor:	Prof. Dr. Martin Bichler
Advisor:	Stefan Heidekrüger
Submission:	14.10.2020

I assure the single-handed composition of this bachelor's thesis only supported by declared resources.

Munich, 14.10.1996

(Daniel Schroter)

Abstract

In serial supply chains, humans tend to show irrational behaviour leading to high cost. The beer distribution game is often taught in management classes to demonstrate the bullwhip effect - a phenomenon describing that orders from the supplier tend to have a higher variance than sales to the buyer. This distortion leads to fluctuations in inventory levels, causing unnecessary high costs. The optimal ordering policy is already known, but unfortunately, humans usually tend to show different behaviour. Hence, we investigate whether reinforcement learning can derive better ordering policies. Thereby the co-players of our intelligent agent either act randomly, optimally or human-like (Sterman formula, 1989). So far, mainly action-value reinforcement learning algorithms have been implemented to solve the beer distribution game. The game is originally defined as a discrete setting (discrete order quantities), which might be the reason that policy-gradient algorithms have not been considered yet. However, in many economic situations, a continuous version of the game is applicable. The ordered quantities, for instance, are so high, that simply rounding them has no major economic impact. Policy-gradient methods have features that can deliver a valuable contribution to research. They can be easily extended from one-dimensional decision making to multi-dimensional decision making. Hence, they can be used to improve supply chains trading multiple items. We create a discrete and continuous game environment that allows a simple experimentation with reinforcement learning algorithms. We further implement the policy-gradient methods REINFORCE and Deep Deterministic Policy Gradient (DDPG). Within this study, we show that they perform on the same level as the current state of the art action-value approaches. Thereby we create a starting point for future research with policy-gradient algorithms in serial supply chains.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AO	=	Arriving order
ANN	=	Artificial Neural Network
AS	=	Arriving Shipments
BDG	=	Beer Distribution Game
bs	=	Base stock
Dec-POMDP	=	Decentralised partially observable Markov decision processes
DDPG	=	Deep Deterministic Policy Gradient
DQN	=	Deep Q Network
IL	=	Inventory Level
MDP	=	Markov decision processes
OO	=	On-order items
POMDP	=	Partially observable Markov decision processes
RF	=	REINFORCE
RL	=	Reinforcement Learning
Sterm.	=	Sterman

1 Introduction

The Beer Distribution Game (BDG) simulates a serial supply chain consisting out of a manufacturer, a wholesaler, a distributor, and a retailer. Each has local information about incoming orders and can order beer from his immediate upstream neighbour in the supply chain. The game is often taught in management classes as it triggers the Bullwhip effect, a phenomenon describing that the stocks of the actors have high fluctuations leading to unnecessarily high costs. It is known that a base-stock-policy would lead to a cost minimum (Clark and Scarf, 1960). However, due to batch ordering, discounts and other reasons, managers tend to show different behaviour leading to inefficiencies.

Reinforcement Learning (RL) has gained much interest through its advancements in playing games like Atari and GO. It can be generally applied to settings where a learning agent interacts with an environment to achieve a goal (Sutton and Barto, 2018, p. 2). The beer distribution game is such a setting, which leads to the question of whether RL can find good ordering policies resulting in efficiency gains. Within this study, we replace the wholesaler with different RL algorithms and let them play with co-players that act randomly, optimally, or human-like (Sterman formula, 1989).

Reinforcement Learning incorporates several classes of algorithms. So far, mainly action-value methods have been implemented to solve the BDG. Oroojlooyjadid et al. (2017), for instance, implement a DQN algorithm that identifies efficient ordering policies. The original BDG is formulated as a game with discrete action space (discrete order quantities). This might be a reason why the class of policy gradient algorithms has not been considered for solving the BDG yet. However, in many economic situations, a continuous version of the game is applicable. For instance, if the ordered quantities are so high, that simply rounding them has no major economic impact. Even if we do not want to deal with real-valued order quantities and incorporate rounding into the continuous policy-gradient methods, their training process seems to get delayed but not prevented. On the other side, policy-gradient methods incorporate a variety of features that can deliver a valuable contribution to the research. They can be easily extended from one-dimensional decision making to multi-dimensional decision making. Hence, they can be used to investigate supply chains trading multiple items.

Within this thesis, we create a framework that allows the simple experimentation with RL-algorithms playing the beer distribution game. Thereby we implement a discrete and continuous game environment and the known policies from the literature. The BDG has some complicating properties such as limited information sharing among the actors during the gameplay. Oroojlooyjadid et al. (2017) introduced some mechanisms to cope with those complicating features. We implement some of their approaches and employ them together with the policy-gradient methods REINFORCE and Deep Deterministic Policy Gradient (DDPG). We conduct several numerical experiments, where our algorithms act in different settings comprising different end-customer demand distributions and co-player policies. We show that they can reduce the cost by up to 77% when replacing the wholesaler in a human-like supply chain with the intelligent agent (DDPG). We further show that they perform on the same level as the current state of the art action-value approaches (DQN). With this thesis, we create an entry point for future research regarding policy gradient methods in serial supply chains.

The study has the following structure: (2) an introduction into the beer distribution game, (3) a brief overview of the literature, (4) a theoretic introduction into the applied reinforcement learning algorithms, (5) the experimental setup, (6) the numerical experiments and performance of the policy-gradient algorithms, (7) a conclusion about the application of policy-gradient algorithms.

2 The Beer Distribution Game

The Beer Distribution Game (BDG) simulates a supply chain consisting out of a manufacturer, a wholesaler, a distributor, and a retailer. Each has local information about incoming orders and can order beer from his immediate upstream neighbour in the supply chain. The game is often taught in management classes as it triggers the Bullwhip effect, a phenomenon describing that the stocks of the actors have high fluctuations leading to unnecessarily high costs for the total

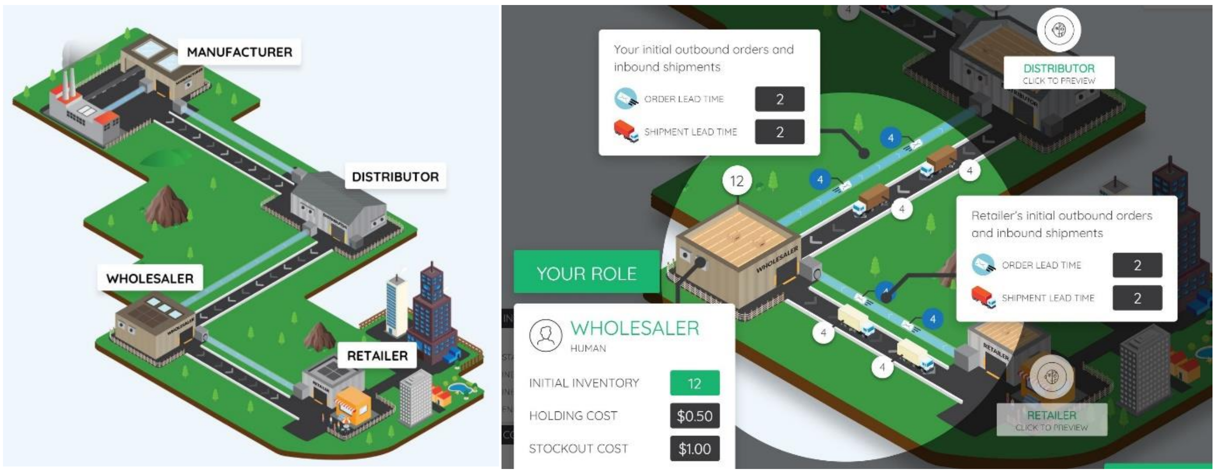


Figure 1: Beer Distribution Game Overview (Retrieved from: https://beergame.opexanalytics.com/#/ , 10.07.2020)

supply chain (Sterman, 1989). We assign the numbers 1 to 4 to the actors representing the supply chain from the retailer to the manufacturer (figure 1). The game is organised in T rounds in which each agent observes the local demand, receives shipments and places replenishment orders q_t^i. The goal of each agent is not to minimise its local cost, but the cost of the total supply chain. Hence the players act cooperatively as a team. The following formula describes the total cost:

$$TC = \sum_{t=1}^{T} \sum_{i=1}^{4} c_h^i * \max\{0, IL_t^i\} + c_s^i * \max\{0, -IL_t^i\} \qquad (1)$$

The formula accumulates over all rounds and all agents the costs that occur concerning the inventory level IL_t^i. So for each agent i we specify the holding cost c_h^i and stockout cost c_s^i. The inventory level can be positive or negative. If it is negative, there are backorders, and stockout cost occurs. For positive inventory levels, there are units at hand, and holding cost occurs. The first term represents the holding cost if there is an inventory on hand. The second term represents the stockout cost if there are backlogged items (negative inventory level) (Oroojlooyjadid et al.,2017).

Shipments and Orders do not immediately reach the customer or supplier, respectively. They need some time to be processed and shipped. Hence for each agent, we further specify a shipment lead time l_{ship}^{i} and an order lead time l_{order}^{i}. The shipment lead time describes the delay of shipments from the supplier to the agent. The order lead time specifies the number of rounds that the order is delayed on the way from an agent to its supplier (see figure 1).

The game lasts several rounds. Usually, the players do not know the number of rounds T to avoid horizon effects. Each round requires all agents to carry out five steps (Sterman, 1989):

1. *Receive inventory and move shipments*. The arriving shipment is added to the inventory. Shipments that are travelling from supplier to the agent are moved one step towards the agent. The number of delays between agent i and $i+1$ is defined by l_{ship}^{i}
2. *Fill orders*. Agents examine the arriving order. Orders are filled to the extent that inventory levels permit. Ordered goods that cannot be delivered add up to the backlog (negative inventory level). The amount that must be delivered incorporates the negative inventory level, if any, and the incoming order. The outgoing shipment is placed on the shipment delay of agent $i-1$.
3. *Inventory level is updated*
4. *Move orders in the order delay*. Inbound orders that are ordered from agent i by customer $i+1$ are moved one step towards agent i. The number of delays is defined by l_{order}^{i}
5. *Place orders*. Each agent decides how much to order q_{t}^{i} and places the order into the order delay.

The retailer and manufacturer need some side notes, as they represent the tails of our supply chain. There is an external demand distribution that simulates the orders arriving at the retailer. The manufacturer has no supplier, but he can produce the item. So instead of a shipment delay, it can also be called production delay describing the time needed to produce an order. Only step five requires a decision to be made by the agent (Sterman, 1989). The agents do not share information during the game. Only after the game has finished, the agents know the total cost that occurred. In this thesis, we will examine whether an intelligent agent will find a way to reduce cost.

3 Literature Review

3.1 Current State of the Art

The beer game is a serial supply chain network and research dealing with such networks is closely related to the beer game setting. There is extensive research covering various aspects of this setting. Such aspects include modelling human behaviour, investigating the role of communication or designing models to derive optimal ordering policies. In this thesis, we implement the BDG and reinforcement learning algorithms to find good ordering policies. Therefore, we mainly focus on research coping with models that deal with ordering policies. However, Martinez-Moyano et al. (2014) provide an overview of the history of the BDG and its rule changes over time. Over the years several variants of the Beer Distribution Game evolved. In this thesis, we rely on the version defined by Sterman (1989) and refer to it as the classical BDG setting in the following. The strategy that leads to minimal cost in this setting is already known. It is called the base-stock policy (Chen, 1999) and only results in minimal cost if applied by all players.

Within the base-stock policy, each agent has a certain inventory-level, called the base-stock level. Further, there is a value called the installation stock. It comprises the on-hand inventory minus the backlogged orders plus the outstanding orders. The agent chooses an order quantity to keep its installation stock equal to the base-stock level. The calculation of optimal base-stock levels is a non-trivial question. Clark and Scarf (1960) suggest a way to calculate the optimal base-stock levels under certain assumptions such as random customer demand and stockout cost at the retailer. Our classical BDG does not fulfil those assumptions. Oroojlooyjadid et al. (2017) use a heuristic approach similar to the methods suggested in Graves (1985) to choose the base-stock levels. We will rely on the values for the base-stock levels they used within their study.

However, due to incomplete information, batch ordering, discounts and other reasons, managers tend to show different behaviour leading to the “Bullwhip effect” (Lee et al., 1997). The bullwhip effect describes a phenomenon where orders to the supplier tend to have a larger variance than sales to the buyer. This distortion propagates upstream in an amplified form (Lee et al., 2004). As a result, there are high fluctuations in inventory levels leading to unnecessary high cost. Lee et al. (1997) and Sterman (1989) examine some of the rational and behavioural reasons. There are several approaches to solve the Bullwhip problem. Wu and Katok (2006), for instance, investigate how communication affects performance during the game. Ponte et al. (2016) investigate methods for profit allocation and corresponding incentives. Sterman (1989) introduced a formula that reflects how human-players over or under-react when they observe large shortages or excess inventory. Note that he does not focus on deriving an optimal ordering policy but instead aims at modelling human behaviour. Therefore, his research is crucial to our study. To decide whether an algorithm should be implemented, it might be valuable to check whether it performs better than humans. Hence his formula will serve as an essential baseline of performance. There are some extensions to the Sterman formula. Strozzi et al. (2007), for instance, use a genetic algorithm to generate the coefficients of the Sterman formula.

For cooperative games, Claus and Boutellier (1998) differentiate between independent learners and joint action learners. Independent learners have no information about the state of the other players, whereas joint action learners share information about their states. The BDG does not allow information sharing during the game. Hence the actors can be classified as independent learners. Claus and Boutellier (1998) further investigate the behaviour of reinforcement learning algorithms in simple multi-agent games and their convergence to certain equilibria. We will see that under certain assumptions, our game converges into a stable state. If the actors follow the base-stock policy and the end-customer demand is distributed as defined by sterman (1989) the inventory levels of our players converge to 0. They will further just be ordering the arriving demands. Every other action would increase their local inventory and subsequently, the local and total cost. Hence there is no incentive for the agents to change their behaviour.

Oroojlooyjadid et al. (2017) review some of the algorithms that seek to derive good ordering policies. Kimbrough et al. (2002) implement a genetic algorithm to search for optimal ordering rules. The rules they formulate are of shape $d + x$ where each agent observes the local demand d and chooses x such that the sum is the ordered quantity. Giannoccaro and Pontrandolfo (2000) and Chaharsooghi et al. (2008) use RL to solve the BDG. Their state variables are the inventory positions of the agents, discretised into 9 and 10 intervals, respectively. Both papers assume information sharing across the agents to simplify the problem. This is why Oroojlooyjadid et al. (2017) identify a gap in academic research. To close this gap, they implement an action-value algorithm (DQN) to solve the classical beer distribution game. To avoid information sharing, they suggest a feedback scheme as a communication framework. In their setting the algorithm only controls one agent meanwhile the other agents are controlled by simple formulas (i.e. Sterman & base-stock) or by human players. We will be closely following their approach throughout this thesis. However, instead of applying an action-value RL algorithm such as DQN, we will transfer some of their ideas to the family of policy-gradient RL algorithms.

3.2 Our Contribution

We create a software environment that facilitates the examination of RL algorithms within the beer distribution game. Our implementation offers a variety of different scenarios and can be configured as a discrete or a continuous version of the game. Both versions differ in a way that the continuous version allows real-valued order quantities. We further include different demand patterns and methods to simulate the behaviour of the agents. For every agent, we may decide whether we want him to be the intelligent learning agent or whether he should act optimally (base-stock), randomly or human-like (Sterman, 1989).

Merely replacing the agents with RL algorithms does not solve the problem. Some traits like the decentralised decision making, cooperative goal and lack of information sharing complicate the problem (Claus and Boutilier, 1998). To address those issues, Oroojlooyjadid et al. (2017) introduced some mechanisms that allow a DQN-algorithm to solve the classical BDG. They, for instance, introduced a feedback scheme that enables the training of the DQN algorithm, although there is no information sharing during the game. Those mechanisms are reimplemented and adopted to allow a respective training of policy-gradient algorithms.

Policy-gradient algorithms can comfortably handle continuous game settings. At first sight, it might be surprising to implement a continuous variant of the game, as we are not able to order fractions of goods in economic reality. Nevertheless, in many economic settings, the amount of ordered goods is so high, that simply rounding the ordered numbers at the end should not have a major economic impact. Incorporating the rounding directly into the algorithm did not lead to significant performance losses, although it increases the training time required (Appendix D). Policy-gradient methods are not just easily applicable to the continuous setting but also offer other advantages. They can be easily extended from one-dimensional decision making (ordering beer) to multi-dimensional decision making (i.e. ordering beer and soft drinks). This study should deliver an entry point and experimental framework for such future research question. Policy-gradient methods have not been considered for the BDG so far and should, therefore, be examined for the problem at hand. Within this thesis, we implement two policy-gradient algorithms. The REINFORCE algorithm (Sutton and Barto, 2018, p. 326) is applied to the deterministic and the continuous version of the game, and the Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al.,2015) is applied to the continuous variant. We will further compare the performance of the algorithms to the action-value method introduced by Oroojlooyjadid et al. (2017)

4 Theoretic Introduction

4.1 Introduction to Reinforcement Learning

Reinforcement learning (RL) is about learning what action to take in a particular situation to maximise a specific reward (Sutton and Barto, 2018, p. 3). Thereby a software agent interacts with an environment and tries to discover behaviour leading to a specific goal. This goal is reflected by rewards that the agent can get for his actions. It usually tries to maximise rewards. For favourable actions, he receives a higher reward. One characteristic of RL is that the learner is not guided towards the desired behaviour. It has to discover actions leading to higher rewards by trial and error. This leads us to another key feature of RL, the exploration-exploitation dilemma (Sutton and Barto, 2018, p. 1f). To receive a high reward, an agent should choose actions which turned out to be successful in the past (exploit). To identify those actions, it must try new behaviour. In other words, it must explore its opportunities. Neither exploration nor exploitation can be pursued exclusively without failing the task. If exploitation is done too extensively, the agent just misses valuable actions. If the agent focuses too much on exploring the world, he does not maximise its rewards representing the proper goal (Sutton and Barto, 2018, p 3). However, it is a crucial characteristic of RL, and we will see how the different algorithms address this issue.

Another typical feature of RL is the delay between actions and their rewards. Some actions might not just influence the immediate rewards but also the following situation and hence all subsequent rewards. So, there might be a delay between actions and rewards (Sutton and Barto, 2018, p. 1). RL is especially applicable when we have an agent interacting with an environment. This usually involves sequential decision making, as the agent is constantly facing new situations and acting upon them. This makes RL an area of machine learning that is especially suitable for sequential decision making (Sutton and Barto, 2018, p. 47). Typical fields of application include games, robotics or business management (Li, 2017).

Those sequential decision processes can be formalised by the concept of Markov Decision Processes (MDP). We consider an agent interacting with an environment (see figure 2). In each time step t the agent observes the current state $S_t \in \boldsymbol{S}$ of the environment and decides which action $A_t \in \boldsymbol{A(s)}$ to take (The set of possible actions depends on the current state). The environment transitions into the next state S_{t+1}. As a consequence of its action, the agent receives the reward $R_{t+1} \in \boldsymbol{R} \subseteq \mathbb{R}$.

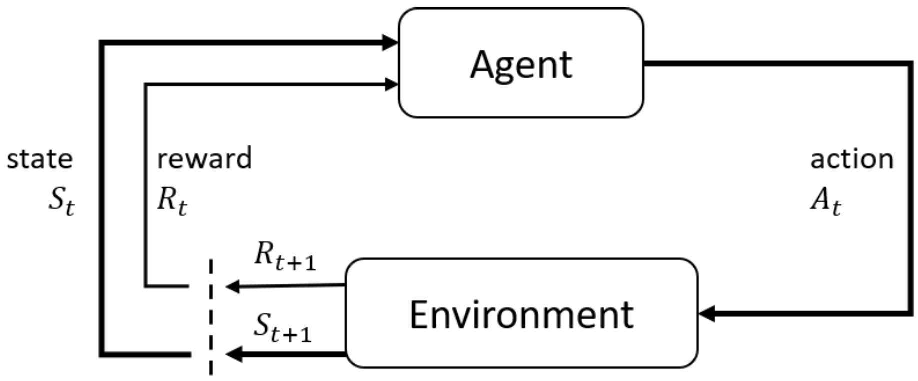


Figure 2: The agent-environment interaction in an MDP, Sutton and Barto (2018, p.48)

Where $\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}$ are the sets of all possible states, actions and rewards, and they are usually finite. Through the interaction with the environment the agent creates a trajectory of states, actions and rewards: $S_0\, A_0 R_1 S_1 A_1 R_2 S_2 \ldots\, A_{T-1} R_T S_T$. Our beer game is an episodic setting. This means that there is a natural end of a trajectory (number of rounds played per game). Afterwards, the game starts again, which is independent of how the previous game ended (Sutton and Barto, 2018, p. 54). At each state during the interaction, we can calculate the probability to transition into the next state and observe a particular reward. The probability of getting into the state s' and receive reward r if the previous state and the corresponding action are given can be calculated by (Sutton and Barto, 2018, p. 48):

$$p(\, s', r \mid s, a) = Pr(\, S_t = s', R_t \mid S_{t-1} = s, A_{t-1} = a) \tag{2}$$

The probability for each combination of S_t and R_t only depends on the preceding state and action. For each action, the agent receives a reward. We do not want to maximise the reward for a single action but a series of actions. Hence the agent's goal is to maximise the total amount of rewards it receives. This leads us to the definition of the return G_t. The return is a function of the rewards. The simplest case is just the sum of future rewards $G_t = R_{t+1} + R_{t+2} + \cdots + R_T$. Especially in very long-term settings or continuing cases, the reward can quickly get towards infinity. Therefore, the rewards are often discounted γ in a way that the return converges (Sutton and Barto, 2018, p. 54f). The return is then defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \; with\, \gamma \in (0,1] \tag{3}$$

Note that the BDG is episodic and in this study considered as relatively short term. However, we included the discounted version to allow experiments with longer runs of the game and to check whether it influences performance. As we do not know deterministically the rewards that we receive in the future, we define our goal as maximising the expected return. Many RL algorithms further use the estimation of value functions. Those functions indicate "how good" it is for an agent to be in a specific state or to be in a specific state and take a particular action.

The “how good” is related to the expected return. This return depends on the current state and the actions that the agent is going to take in the future. To define value functions, we need a concept describing the way the agent acts in certain situations. This behaviour is formalised within the idea of policies π. In each time step, the agent finds itself in state $S_t = s$ and chooses the action $A_t = a$ according to a particular policy π. Thereby $\pi(a|s) = Pr(A_t = a|S_t = s)$ defines the probability of taking action a given the state s at time-step t. The core of RL algorithms is to change the agent’s policy in a way that a higher return is expected (Sutton and Barto, 2018, p. 54f).

With the definition of the policy, we have the ingredients to define value functions. The state-value function v_π describes the expected return if we consider ourselves in state s and the agent acts according to the policy π until the rest of the game.

$$v_\pi(s) = E[G_t|S_t = s] = E_\pi\left[\sum_{k=0}^{T} \gamma^t R_{t+k+1}|\, S_t = s\right], for\ all\ s\ \in \boldsymbol{S} \tag{4}$$

The action-value function Q_π describes the expected return, if we consider ourselves in state s, take action a and then follow the policy π until the rest of the game.

$$Q_\pi(s,a) = E[G_t|S_t = s, A_t = a] = E_\pi\left[\sum_{k=0}^{T} \gamma^t R_{t+k+1}|\, S_t = s, A_t = a\right] \tag{5}$$

If $p(s', r\ |s, a)$ is given, then optimal policies can theoretically be found through dynamic or linear programming. However, in many practical applications (e.g. the beer game), we deal with large state and action spaces. The methods to find optimal solutions often require more computational power than available. One fundamental property of RL is to approximately solve MDPs (Sutton and Barto, 2018, p. 67f).

The beer game setting has some properties that further complicate the problem. The state variable is not fully accessible to the agent. In other words, the state of the environment can only be partially observed by the agent. This scenario is formalised by partially observable Markov decision processes (POMDP) (Sutton and Barto, 2018, p. 466f). Furthermore, there is an intelligent agent which can only observe partial information but must cooperate in a decentralised manner with multiple other agents to achieve a common goal. Such problems are called Dec-POMDP, and according to Bernstein et al. (2002), the problem is NEXP-complete. There is no polynomial-time algorithm and probably even no exponential-time algorithm that solves the problem. We will investigate whether reinforcement learning can find an approximate solution.

The interested reader recognises that we introduced finite MDP. The state, action and reward spaces are finite sets. This formalisation is applicable when we deal with discrete settings. As already mentioned, we extend the BDG towards a continuous setting. The state and action space incorporate an infinite set of real-valued numbers. The underlying logic of the MDP remains the same. A formalised mathematical description of MDP with infinite sets requires more complex notations without providing valuable information for this study. Hence, we will stick to the discrete notations.

4.2 Policy-gradient algorithms

In Reinforcement Learning, there are two main areas when it comes to approximate solution methods. Action-value methods learn the value of actions and then select the action based on the estimated action value. Q-Learning, for instance, is a method where the action-value function Q is learned. The Deep Q-Learning (DQN) applied by Oroojlooyjadid et al. (2017) is such a method. The policy is indirectly improved by updating the learned Q-function (Sutton and Barto, 2018, p. 131). Because of estimating the value of actions, those methods are hardly applicable to continuous action spaces. On the other hand, there are policy gradient methods that learn a parameterised policy. As we know, the policy defines the behaviour of the agent in certain situations. The policy is determined by parameters, and the parameter vector is notated with θ. The policy is explicitly given by $\pi_\theta(a|s)$. The probability that action a is taken at time t depends on the current state s of the environment and the parameters θ of the policy (Sutton and Barto, 2018, p. 321f). This probability can be calculated using the following formula:

$$\pi_\theta(a|s) = Pr(A_t = a \,|S_t = s, \theta_t = \theta) \tag{6}$$

The fundamental idea is to adjust this probability distribution in a way, that the actions that lead to a higher expected return get a higher probability assigned. However, there are multiple actions with a chance to be taken. This is called a stochastic policy, and an example is the REINFORCE algorithm, we apply in this study. On the other hand, we have deterministic policy gradient methods, such as DDPG, that define the actions deterministically. Hence given a particular state and parameter vector, the algorithm outputs the action to take. In other words, the probability for this action is equal to 1, whereas the probabilities for all other actions are 0. A policy with a parameter vector could, for instance, be a neural network mapping states to actions. In that case, the parameters θ of the policy are the weights of the neural network. Learning then means enhancing the parameters that define the policy, such that a higher return is expected. A value function might still be present in the algorithms, but in this case, it is not used to estimate action values but to learn the policy parameters (Sutton and Barto, 2018, p. 231f).

To improve our policy, we investigate a scalar performance measure $J(\theta)$ for the policy parameter. Performance is maximised by updating the policy with the approximate gradient ascent in J:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)} \tag{7}$$

This means we modify our policy towards the direction, that results in higher expected performance. $\widehat{\nabla J(\theta_t)}$ is not the exact gradient but a stochastic estimate whose expectation approximates the actual gradient. α is a step size parameter, called the learning rate, defining the size of adjustment we apply to our policy (Sutton and Barto, 2018, p. 231f).

In our episodic case, we define performance as the state-value of our start state: $J(\theta) = v_{\pi_\theta}(s_0)$. The performance depends on both the selection of actions and the distribution of states in which those selections are made. Both are influenced by the policy parameter (Sutton and Barto, 2018, p. 324). Hence it gets difficult to change the policy parameter in a way that ensures improvements through a better selection of actions. Employing the policy gradient theorem (Sutton and Barto, 2018, p. 325), the problem can be theoretically reformulated:

$$\nabla J(\theta) = E\left[\sum_{t=0}^{T} \nabla \ln \pi_\theta(a_t|s_t) G_t(\tau)\right], with\ G_t(\tau) = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{8}$$

This is a significant result. We can now approximate our gradient by sampling data with our policy and calculating an expected value. The policy gradient method now has to deal with two main aspects. First, sampling actions according to a particular policy and thereby generating data. And secondly calculating the log probabilities of an action given a specific probability distribution $\pi_\theta(a_t|s_t)$.

4.2.1 REINFORCE

In our policy gradient algorithms, the policy is approximated with an artificial neural network with parameter vector θ. In the REINFORCE algorithms of Sutton and Barto (2018, p. 328f), one episode of the game is played and the policy is updated afterwards. Our version is slightly adjusted because we want to gather more experience before taking an update step. Therefore, the algorithm is modified in a way, that it plays N episodes of the game, creating a set of trajectories $D = \{\tau_i\}_{i=1,\dots,N}$, where τ_i represents one trajectory. Within this batch of games, the agent acts upon policy π_θ. After each game played, we know the total cost that occurred and adjust the rewards for each action according to the feedback scheme suggested by

Oroojlooyjadid et al. (2017). The feedback scheme takes the teamplay of an agent with respect to the total cost into account (section 5.1). Based on the experience generated, we take an update step of our policy. As we can see in formula 8, the gradient is an expectation so that we can estimate it with a sample mean of our created experience (Achaim, 2018, simple policy gradient). To estimate the gradient, we use the set D of played games:

$$\widehat{\nabla J(\theta)} = \frac{1}{N}\sum_{\tau\in D}\sum_{t=0}^{T}\nabla \ln \pi_\theta(a_t|s_t)G_t(\tau)\ , \qquad with\ G_t(\tau) = \sum\nolimits_{k=t+1}^{T}\gamma^{k-t-1}R_k \tag{9}$$

Note that we use $G_t(\tau)$ for the return of a particular trajectory now, because we create multiple trajectories and must introduce a notation to identify those. The pseudocode for our REINFORCE algorithm is shown below. The main questions about sampling actions during the game and calculating the logarithmic-probabilities have not been discussed yet. They differ in between the discrete and continuous variant of the algorithm and will be introduced in the following.

```
      Algorithm 1: REINFORCE: Pseudocode
1:    Input: a differentiable policy parameterisation $\pi_\theta(a|s)$
2:    Algorithm parameter: step size $\alpha$ > 0
3:    Initialise policy parameter $\theta$
4:    Loop forever:
5:        Generate N episodes: $S_0\ A_0R_1S_1A_1R_2S_2$ ... $S_{T-1}A_{T-1}R_T$ , following $\pi_\theta$
6:        Loop for each episode:
7:            Loop for each step of the episode:
8:                $R_t$ = calculate feedback (formula 14)
9:                $G_t(\tau)$ = $\sum_{k=t+1}^{T}\gamma^{k-t-1}R_k$, (calculate discounted rewards to go)
10:       $\theta = \theta + \alpha\frac{1}{N}\sum_{\tau\in D}\sum_{t=0}^{T}\nabla \ln \pi_\theta(a_t|s_t)G_t(\tau)$ (take an update step)
```

4.2.1.1 *REINFORCE for discrete action space*

Within the REINFORCE algorithm for discrete action spaces the policy $\pi_\theta(a|s)$ outputs the probability for each of the possible actions depending on the state and the parameter vector. To be precise, we output the logits for the possible actions. However, this is less important here as it is just a technical thing to avoid giving to many boundaries to the neural network. The interested reader is referred to Fahrmeir et al. (2016, p. 464). Those probabilities then characterise a categorical probability distribution. When the agent interacts with the environment, it uses the probability distribution to sample its action, leading to a new state.

Given the new state, our policy outputs another categorical probability distribution, which again is used to sample the corresponding action. Given a probability distribution, the calculation of logarithmic probabilities is straightforward. By randomly sampling the actions, we automatically ensure that there is some degree of exploration happening because there are several actions with a certain probability Sutton and Barto (2018, p. 328f).

4.2.1.2 REINFORCE with Gaussian policy

For the continuous REINFORCE algorithm, we require the same two steps. Given a specific state, we derive a probability distribution that reflects the likelihood for the actions. We then use that distribution to sample actions. In a continuous action space, we can simply change the derived probability distribution from a categorical probability distribution to a normal distribution. The policy does not longer output the probabilities for the actions but instead the mean $\mu(s,\theta)$ and standard deviation $\sigma(s,\theta)$ that characterise a normal distribution. This approach is called a gaussian policy. Hence $\pi_\theta(a|s)$ is characterised by the density function of a normal (Gaussian) distribution:

$$\pi_\theta(a|s) = \frac{1}{\sigma(s,\theta)\sqrt{2\pi}} \exp\left(-\frac{\big(a-\mu(s,\theta)\big)^2}{2\sigma(s,\theta)^2}\right) \tag{10}$$

Given the density function, we can easily calculate the logarithmic probabilities needed for our gradients. The idea behind the gaussian policy is that the mean specifies the order quantity and the standard deviation ensures the degree of exploration around the mean. As the training evolves the standard deviation should decrease, so we get more precise about which action to take. In this study, we only treat the classical BDG setting, where one item is traded within the supply chain. However, this setting could be extended from a one-dimensional action space to a two-dimensional action space by adding a second pair of mean and standard deviation as output (Sutton and Barto, 2018, p. 357f).

4.2.2 Deep Deterministic Policy Gradient

The Deep Deterministic Policy Gradient (DDGP) is a method that combines advantages from action-value and policy-gradient algorithms. It is only applicable to continuous action spaces. The DDPG algorithm was introduced by Lillicrap et al. (2015). In the following, we rely on the notations given by Achaim (2018, DDPG) in the OpenAI SpinningUp implementation of the DDPG implementation.

The DDPG makes use of the action-value function $Q_\pi^*(s,a)$ that we already introduced in section 4.1. The little star indicates that we talk about the optimal action-value function describing the value of state s if we take action a and follow the optimal policy π^* hereinafter.

Note that we used π to describe stochastic policies. For deterministic policies we use $\mu(s)$ instead. To further simplify notations we will just use $Q^*(s,a)$ for the optimal action-value function in the following. The DDPG significantly differs from the already introduced REINFORCE algorithms in a way that it deterministically derives the action given a certain state. If we know the optimal action-value function $Q^*(s,a)$ then a greedy policy $\mu(s)$ simply derives the optimal action, given a specific state by taking the action with the highest Q-value assigned: $\mu(s) = a^*(s) = arg\, max_a Q^*(s,a)$. Hence our goal is to approximate the optimal action-value function (Achaim, 2018, DDPG).

The starting point for our algorithm is the Bellman equation for the optimal action-value function. The basic idea of the Bellman equation is that the value of a particular state is the reward we expect to get from being there and taking action a, plus the value of wherever we land next.

$$Q^*(s,a) = E_{s' \sim P(*|s,a)}[r(s,a) + \gamma max_{a'} Q^*(s',a')] \tag{11}$$

The $s' \sim P(* \,|s,a)$ indicates that the next state s' is sampled from the environments transition rules. The max represents the fact that when we choose our action, we pick the action that leads to the highest value. As we do not know the optimal bellman function $Q^*(s,a)$, we train a neural network $Q_\delta(s,a)$, with parameters δ, to be an approximator. We further consider a set D of transitions $(s, a\,, r, s', d)$, where d indicates whether s' is a terminal state (d=1). If we reach a terminal state, we cannot expect any additional rewards. During the learning process, the bellman equation plays a vital role because we try to minimise the mean-squared bellman error (MSBE). The MSBE roughly describes how close our approximator Q_δ gets towards the Bellman equation (Achaim, 2018, DDPG).

$$L(\delta, D) = E_{(s,a,r,s',d)\sim D}\left[\left(Q_\delta(s,a) - (r + \gamma(1-d) max_{a'} Q_\delta(s',a')\right)^2\right] \tag{12}$$

The DDPG algorithm incorporates three features that slightly modify the loss function to enhance training and should therefore be briefly introduced. (Achaim, 2018, DDPG).

1. *Replay Buffers*: In the DDPG, there is a storage for transitions. We call it replay-buffer D. When interacting with the environment, new transitions are added to the replay-buffer. When it comes to learning, we take a small sample of transitions out of the buffer and train our networks. This is done because many optimisation algorithms assume independent and identically distributed samples. If we would take samples out of a single run of the game, they are correlated because a particular state depends on the previous ones (Achaim, 2018, DDPG).

2. *Target Networks:* The later term of the MSBE loss function is called the target: $r + \gamma(1-d)max_{a'}Q_\delta(s', a')$. When we are minimising the MSBE loss, we try to approximate the target with our Q-function. The target depends on the same parameters δ that are trained, which leads to instabilities during the MSBE minimisation. As a solution, we make use of a second network, the target network, which is a copy of the main network. It uses a set of parameters δ_{targ} which come close to δ but with a time delay. In the DDPG algorithms, the target network is updated once per main network update using Polyak averaging (Polyak, 1990): $\delta_{targ} = \tau\delta_{targ} + (1-\tau)\delta$, with $\tau \in (0,1)$, but usually close to 0. The weights of the target networks slowly track the learned network (Lillicrap et al., 2015)).
3. *Maximum over continuous actions:* We consider ourselves in a continuous action space. Hence the calculation of the maximum over actions $max_a Q(s,a)$ is an expensive subroutine because there is an infinite number of possible actions. As our action space is continuous, we can assume a differentiable Q-function with respect to the actions. This allows us to use a gradient-based learning rule for the policy $\mu(s)$. If we have a differentiable Q-function, we can approximate $max_a Q(s,a)$ with $Q\big(s,\mu(s)\big)$. To compute an action that approximately maximises $Q_{\delta_{targ}}$ the DDPG implements a target policy network $\mu_{\theta_{targ}}$. The target policy is learned the same way as the target Q-function: by Polyak averaging the policy parameters during training (Achaim, 2018, DDPG).

With those modifications, our MSBE loss function slightly changes towards:

$$L(\delta, D) = E_{(s,a,r,s',d)\sim D}\left[\left(Q_\delta(s,a) - (r + \gamma(1-d)Q_{\delta_{targ}}\left(s', \mu_{\theta_{targ}}(s')\right)\right)^2\right] \qquad (13)$$

Although the DDPG algorithm uses methods from Q-Learning, it remains a policy gradient algorithm. We want to learn a policy $\mu_\theta(s)$ that deterministically gives the action that maximises $Q_\delta(s,a)$. Hence concerning the policy parameters, we can apply gradient ascent to solve: $max_\theta E_{s\sim D}[Q_\delta(s, \mu_\theta(s))]$ (Achaim, 2018, DDPG).

As our policy deterministically derives actions, we have to consider the exploration/exploitation dilemma. During training, we artificially create noise that slightly changes the derived actions to introduce exploration. As suggested in the DDPG paper of Lillicrap et al. (2015), we use the Ornstein-Uhlenbeck noise (Uhlenbeck and Ornstein, 1930).

```
     Algorithm 2: Deep Deterministic Policy Gradient: Pseudocode
1:   Input: initial policy parameters $\theta$, Q-function parameters $\delta$, empty replay buffer $D$
2:   Set target parameters equal to the main parameters $\theta_{targ} = \theta, \delta_{targ} = \delta$
3:   **Repeat**:
4:           Observe state s and select action a = clip($\mu_\theta(s) + \varepsilon, a_{low}, a_{high}$), $\varepsilon \sim Noise$
5:           Execute $a$ in the environment
6:           Observe next state $s'$, reward $r$, and done signal $d$ and save transition $(s, a, r, s', d)$ in
             temporary episode storage
7:           If $s'$ is terminal, apply feedback scheme (formula 14) to transitions in episode storage,
             reset environment state
8:           Add transitions in episode storage to replay Buffer D and reset episode storage

9:           **if** it is time to update **then**
10:                  Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $D$
11:                  Compute targets: $y(r, s', d) = r + \gamma(1-d)Q_{\delta_{targ}}(s', \mu_{\theta_{targ}}(s'))$
12:                  Update Q-function by one step of gradient descent using:
                         $$\nabla_\delta \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_\delta(s,a) - y(r,s',d))^2$$
13:                  Update policy by one step of gradient ascent using
                         $$\nabla_\theta \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} Q_\delta(s, \mu_\theta(s))$$
14:                  Update target networks
                         $$\delta_{targ} = \tau\delta_{targ} + (1-\tau)\delta$$
                         $$\theta_{targ} = \tau\theta_{targ} + (1-\tau)\theta$$
             end **if**
     **until** convergence
```

The pseudocode of algorithm 2 is basically the DDPG algorithm given by Lillicrap et al. (2015) and implemented by Achaim (2018, DDPG). They take a learning step after every single interaction with the environment. Hence this algorithm is especially applicable to continuing problems, where no clear end of an episode exists. In our episodic case, such an end exists. Even more important is that our agents only know the total cost of the game after it ends. To include this information into the design of the rewards, we have to store the transitions of one episode of the game into an episode storage and wait until the episode is over. Afterwards, we adjust the rewards with the feedback scheme and finally add all transitions of the episode storage to the replay buffer. As the replay buffer is usually large (i.e. 1.000.000 transitions) and the samples used for training are typically small (i.e. 32 transitions), we assume that our adjustments do not have a significant influence on the performance. It is not likely that they would have been chosen out of the replay buffer. We are not going to prove that formally, but the learning progress seems to justify the assumption.

5 Experimental setup

5.1 Algorithmic key components

We must define some key variables to create an environment that supports a reasonable interaction with the intelligent agent. For instance, we must control the available information at each agent, the design of rewards and architecture of our neural networks. Therefore, we briefly introduce some of the critical components.

State variables: Each agent observes four variables. Consider agent i at time t. The first variable is the inventory level IL_t^i that we already introduced. The second variable is the on-order items OO_t^i. It describes the items that have been ordered from agent $i+1$ but not received yet. The arriving order AO_t^i describes the local demand reaching agent i from its customer $i-1$. Downstream the supply chain the agent receives shipments AS_t^i from his immediate supplier $i+1$. The variables AO_t^1 and AS_t^4 need special attention as they represent the tails of the supply chain. The first variable describes the end-customer demand, whereas the latter describes the production of the beer. In each period, every agent observes those four variables. Hence in period t agent i has the historical observations $o_t^i = [(IL_0^i, OO_0^i, AO_0^i, AS_0^i), \dots, (IL_t^i, OO_t^i, AO_t^i, AS_t^i)]$. With our assumption of no information sharing, we model the BDG as POMDP. In other words, each agent can only access its locally observable state variables. However, after each time step the observation vector o_t^i grows. We are going to use neural networks to predict the probabilities for taking a particular action. It is inconvenient to handle variable input sizes for neural networks, so we will only use the m last observations as the state variable. Concluding the state variable comprises $S_t^i = [(IL_{t-m+1}^i, OO_{t-m+1}^i, AO_{t-m+1}^i, AS_{t-m+1}^i), \dots, (IL_t^i, OO_t^i, AO_t^i, AS_t^i)]$ (Oroojlooyjadid et al. (2017)).

The ANN architecture: There are many possibilities to design the structure of an artificial neural network (ANN). As input, we use our state variable. The outputs depend on the algorithm we are using. Within the discrete REINFORCE, it outputs the logistic probabilities for all possible actions. Within continuous REINFORCE, it outputs the mean and standard deviation that characterise a Gaussian policy. We experimented with different amounts and sizes of hidden layers and used two hidden layers of sizer 32 in the end. The DDPG algorithm uses two different neural network architectures. One for training the policy and one for estimating the Q-function. Lillicrap et al. (2015) precisely describe both network architectures.

Action Space: We consider both the discrete and continuous BDG. Our end-customer demand gets up to eight items per period. Hence, we used an action space reaching from 0 to 16. For the discrete BDG, we force the actions to be natural numbers. For the continuous version, real-valued orders are allowed. Further increasing the action space did not lead to better results.

Reward Function: Consider an agent i at time t taking action A_t. We have to think about how we can assign a reasonable reward value R_t^i to his actions. After acting the state transitions from S_t^i to S_{t+1}^i. We can subsequently calculate the new inventory level IL_{t+1}^i and further the corresponding shortage and holding costs. The sum of both is considered as R_t^i. In our setting, we have transportation and order lead times. Hence the cost that occur in a certain period are not directly related to the action taken in the previous period, but instead is a result of the actions taken in prior periods. Nevertheless, as we defined our state variable to include the last m periods, we still retain some information about the previously taken actions. Optimising for the rewards R_t^i would lead to a minimisation of the local cost occurring at the agent. The local cost minimisation would not necessarily lead to a minimum regarding the total cost of the entire supply chain. The total cost is defined by $\sum_{i=1}^{4}\sum_{t=1}^{T} R_t^i$. The information about the cost of the other agents is only shared after the game ends. To add this information into the design of the rewards, we consider the feedback scheme of Oroojlooyjadid et al. (2017).

Feedback Scheme (Oroojlooyjadid et al., (2017): After the game ends, the information about the total cost (total reward) is shared among the agents. The goal of our agent is to minimise the total cost. Therefore we must include this information into the rewards the agent receives for its actions. After one episode of the game is played, we update the rewards with the formula suggested by Oroojlooyjadid et al. (2017):

$$R_t^i = R_t^i + \frac{\beta_i}{3}\left(\omega - \vartheta^i\right) \tag{14}$$

Where $\vartheta^i = \frac{1}{T}\sum_{t=1}^{T} R_t^i$ is the average reward per time-step of agent i and $\omega = \sum_{i=1}^{4}\vartheta^i$ is the average reward per period. The factor β_i is a regularisation coefficient. If it is too low, our agent tries to minimise its own local cost. If it is too high, the focus lies on cost minimisation of the co-players. Based on a grid search we choose β_i = 100, as it seems to deliver good results. The role and optimisation of this parameter should be discussed in future research.

Determining the value of m: As already mentioned the inclusion of the m past time steps is crucial to include valuable information about previous states and hence for the design of rewards. Thereby m is related to the order and shipment lead times. Oroojlooyjadid et al. (2017)

suggest that m should ideally be chosen as large as the highest possible delay. This delay is the sum of all shipment and order lead times: $\sum_{j=1}^{4}(l_{ship}^{j} + l_{order}^{j})$. It is the time one order requires to propagate from the retailer up the supply chain, getting produced and shipped down the supply chain to the retailer again. However, a large m increases the size of our neural network and hence slows down the training process. The selection of m results in a trade-off of accuracy and computational resources. Ideally, it would be 15 in our setting, but due to limited computational resources, we use m = 5. When using undiscounted rewards, increasing m only led to small performance improvements (Appendix A).

5.2 Our experimental framework

One part of this thesis is to create a setting for further research regarding the beer distribution game. To facilitate the use of the environment, it seems reasonable to meet some standardised requirements. Several open-source frameworks facilitate the application of reinforcement learning. The probably most used ones are part of OpenAI. With the Gym toolkit, OpenAI standardises the formulation of environments in which RL-Agents can be trained. Furthermore, many open-source frameworks that develop RL-algorithms can be easily tested with environments implementing the Gym interface. Hence our BDG will meet those requirements.

For the implementation of the modules, we used open-source libraries, tutorials, and other publicly available code. For the BDG environment, we got inspired by Orlov (2019). For the REINFORCE agents, we followed the OpenAI SpinningUp implementation of the simple policy gradient algorithm (Achaim, 2018, simple_pg.py). The DDPG algorithm is closely related to the implementation of Tabor (2019, ddpg_torch.py). For the Sterman formula, we used the R-implementation of Edali and Yasarcan (2014). We implemented the framework in python 3.7.7 with PyTorch 1.4.0. The code runs on a local machine with a total RAM of 16 GB and six cores, each with 2.20 GHz.

We create a modular software framework that allows the flexible experimentation with various agents playing the game (see figure 4). Therefore, we identified four main components that should work independently: (i) the BDG environment, (ii) the main training routine, (iii) a class that simulates the behaviour of known policies and (iv) the intelligent agents. The BDG environment includes the logic of the game and implements the OpenAI Gym interface. The main script steps through the environment by passing the actions to the environment and receiving the new states and rewards. It handles the batch creation and storage of trajectories. The environment expects one action for each agent. Hence, we can flexibly decide how our agents should behave. Their behaviour can be controlled by one of the known policies from literature or by one of the intelligent agents. For the intelligent agents, we created separate classes that handle the learning procedure, loss calculation and action sampling according to their current policy. Further, we implement an agent simulator class that includes a method to derive actions according to the optimal base stock policy, random policy or sterman formula.

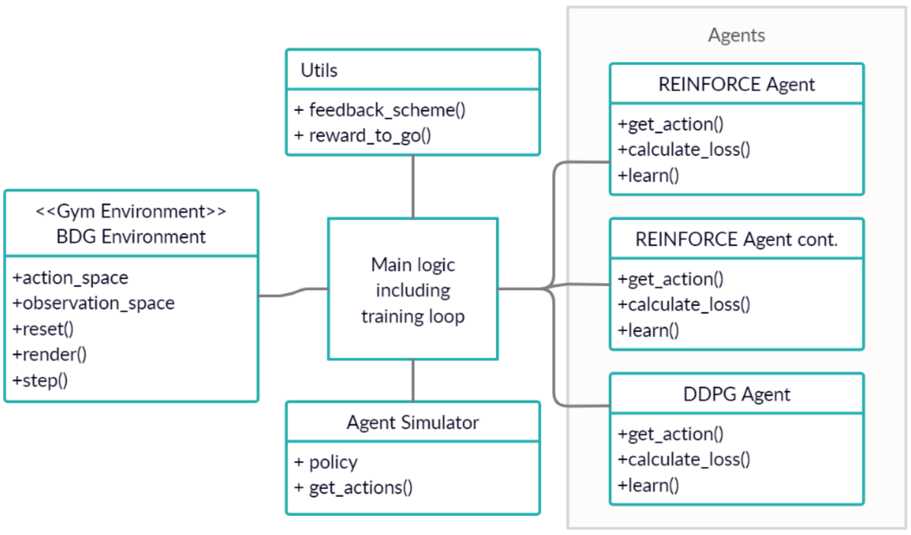


Figure 3: Software Architecture of the Experimental Framework

We implemented some unit tests to ensure whether our logic is correct. To see if our BDG works correctly, we compare the total cost that occur to known values from the literature. For the base stock policy, we rely on values given in Oroojlooyjadid et al. (2017). For the simulation of human-like behaviour, we use values from Edali and Yasarcan (2014).

5.3 Training and Evaluation process

To understand how we conduct numerical experiments, we briefly describe the training and evaluation process. We describe the scenario, where an RL algorithm controls the wholesaler and the co-players follow a known policy from research. The logic for the REINFORCE and DDPG algorithms is similar. We refer to them as intelligent algorithms in the following.

Training procedure: Within the main logic, we create our BDG environment and specify it as discrete or continuous. In our implementation, we assigned the values 0 to 3 to the four agents of our supply chain. We determine the index of the agent that is controlled by the RL-algorithm. In our case, it is equal to 1 as the intelligent agent should play the wholesaler. With this index, we can now request the action and observation space for our agent from the BDG environment. We further create our intelligent agent and pass the possible action space to him. It is usually 16 representing orders reaching from 0…15 with natural numbers for the discrete and real-valued numbers for the continuous case. To define the behaviour of the other agents, we create an agent simulator class and specify the policy of the co-players.

The training loop slightly differs for the REINFORCE and DDPG algorithm. For the REINFORCE algorithm, each iteration of the training loop (epoch) comprises two steps. We first play the game to gather experience, and we secondly update the policy, which represents a learning step. To gather experience, we play a bunch of games (batch) where the intelligent acts upon its current policy. The agent simulator calculates the actions of the co-players. After each game, we adjust the reward of the agent with the feedback scheme (formula 14). We further calculate the rewards-to-go (formula 3). When we have played some games and gathered a certain amount of transitions (batch size), we take a learning step. The data from our batch is passed to the REINFORCE agent to take one update step of his policy. This process is repeatedly done until performance converges.

The procedure for the DDPG algorithm is slightly different because it uses a replay buffer. Instead of gathering a certain amount of transitions, it plays only one game, applies the feedback scheme and adds the transitions to the replay buffer. The DDPG takes a learning step after every single step within the environment by sampling data out of the replay buffer. Hence the learning steps are independent of the game endings.

Evaluation procedure: To evaluate the performance, we simply use our trained agents and let them play 1000 rounds of the game. We use the mean of the total cost of the games as an evaluation metric. We use the optimal base-stock policy by Clark and Scarf (1960), as an upper performance baseline. We use the formula of Sterman (1989) to compare the performance of our agent with the performance of simulated human behaviour. The human-like behaviour plays a crucial role because when we assume that humans currently manage a supply chain, we are interested in the cost reduction if we implement an intelligent agent.

6 Numerical Results

Our framework allows each actor to follow its own policy. This leads to a variety of possible policy constellations. Due to computational limitations, we only investigate the case where one agent is a learning agent and all its co-players follow the same co-player policy. The learning agent plays the role of the wholesaler. However, those experiments can be easily done for the other agents or extended to multiple intelligent agents. We consider three types of policies for the co-players: (i) a random policy, (ii) human-like (Sterman, 1989) and (iii) the base stock policy.

To identify the different combinations, we introduce the notation of shape **wholesaler policy-co-player policy**. For instance, **DDPG-sterman** represents the setup in which the DDPG algorithm controls the wholesaler, whereas the co-players act upon the sterman formula.

We conduct the experiments for two demand distribution:

1. C(4,8) = In the classical demand distribution of Sterman (1989), the end-customer orders 4 items in the first four periods and continuously 8 items afterwards.
2. U[0,8] = The demand is uniformly distributed between 0 and 8

We usually assume that our agents do not have information about the end-customer demand. Anyhow, when the co-workers act randomly, we introduce some knowledge about the demand distribution into the random policy. When we apply the classical demand distribution, the randomly acting co-workers draw their actions uniformly from 0 to 15. When we investigate the uniformly distributed demand, the randomly acting co-workers draw their actions from the same distribution. Hence the orders are also uniformly distributed between 0 and 8. Enlarging the upper boundary would increase the cost because the agents now regularly place orders that exceed the possible end-customer demand. As a consequence, this leads to higher inventory levels and holding cost. To avoid that the cost is artificially increased to make the algorithms look better, we decided to infer the knowledge about the demand distribution. This information only affects the randomly acting co-works, not the intelligent agents. They use their action space as defined in section 5.1 and find reasonable policies on their own.

Demand	c_h	c_s	BS level
C(4,8)	[0.5, 0.5, 0.5, 0.5]	[1, 1, 1, 1]	[32, 32, 32, 24]
U[0,8]	[0.5, 0.5, 0.5, 0.5]	[1, 1, 1, 1]	[19, 20, 20, 14]

Table 1: Cost structure and base stock levels

Furthermore, the optimal base stock levels depend on the demand distribution. Cost parameters and base-stock levels for both distributions are shown in table 1. The calculation of optimal base stock levels is a non-trivial question. We refer to Clark and Scarf (1960) and Chen et al. (1999) for further information. Within this study, we rely on values that have been previously used for those demand distributions (Oroojlooyjadid et al., 2017).

The BDG environment and the algorithms themselves include a variety of variables. Appendix B contains a full list regarding the parameter selection. Those variables can directly influence performance. Some of the key variables are introduced in section 5.1. We conducted a grid search to evaluate certain parameter combinations to find out a promising setup for our numerical experiments. The feedback mechanism includes a regularisation coefficient β. According to Oroojlooyjadid et al. (2017), there is no simple rule to derive a value for β. Hence we tried two values (10 and 100) for beta within the grid search. We further included the number of observed periods m (5 and 10) and the reward discount factor γ (1 and 0.95). We conducted the grid search for the discrete version of the BDG under the classical demand distribution. We choose $\beta = 100, m = 5\ and\ \gamma = 1$. A larger m leads to better results, as more information is included in the decision process. We still use m = 5 because a larger m slows down the training process. For different setups, there might be a better selection of those parameters, which is subject of discussion. Appendix A shows the results of the grid search.

In the subsequent section, we take a look at the performance of our algorithms. We first consider the case where our REINFORCE algorithm plays the discrete version of the game and then have a closer look at the continuous variants.

6.1 Discrete REINFORCE algorithm

We trained the discrete REINFORCE agent for 1000 epochs under the classical and 1200 epochs under the uniform demand distribution. Although there still might be room for improvement, we stopped training as we did not expect any major performance steps. We used a batch size of 5000 and each game ended after 36 rounds. Hence one epoch includes 139 ($\lceil 5000/36 \rceil = 139$) games played. Training for 1000 epochs contains the experience of 139 000 games. For the neural network, we used the Adam optimiser with a learning rate of 0.0001.

Under the classical demand distribution, we conducted three experiments with the discrete REINFORCE algorithm. The wholesaler is played by the intelligent agent and his co-workers acting upon one of the previously mentioned policies. Figure 5 displays the training process of the REINFORCE-sterman setup. We see how the training process converges.

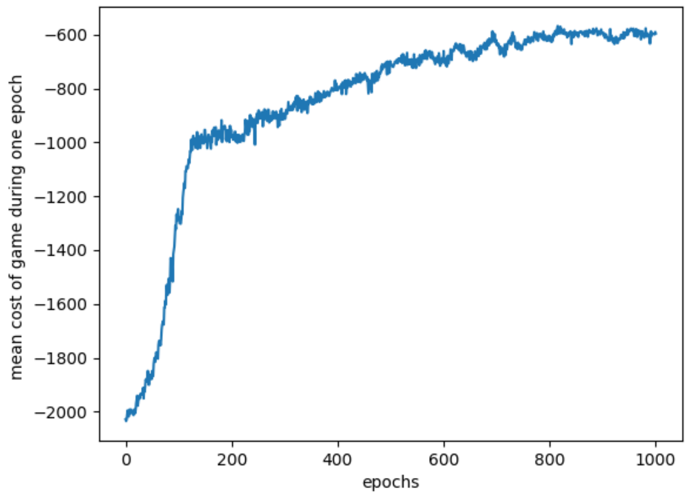


Figure 4 Training process of the REINFORCE-sterman setup

Table 2 shows the results after training the agent for 1000 epochs. The rows indicate whether the wholesaler was controlled by a known policy or our intelligent REINFORCE Agent. The columns represent the behaviour of the co-workers. The base stock policy is optimal if all players act according to it. Therefore, the cost of 180 is a lower bound. Note that under a classical demand distribution, the system gets into a Nash equilibrium. All inventory levels are 0, and each agent only orders the locally observed demand. No agent has an incentive to show different behaviour because this would subsequently lead to higher local and higher total cost. The system remains at total expenses of 180 regardless of the length of the game. Hence for longer games, it gets harder for our algorithms to approach the lower bound.

	Co-players policy					
	classical			uniform		
wholesaler policy	random	base stock	sterman	random	base stock	sterman
random	1.900,2	1.668,4	1.426,1	1135,9	808,8	892,8
base_stock	3.037,9	180,0	3.664,5	1119,3	677,7	1172,4
Sterman	1.831,2	2.865,0	2.049,0	1130,9	962,6	1058,8
Rf_discrete	1.755,5	415,5	594,4	1148,4	804,2	871,1
% Gap to Sterm.	-4,1%	-85,5%	-71,0%	1,6%	-16,5%	-17,7%

Table 2: Performance of the discrete REINFORCE algorithm

However, if the co-players act according to a base-stock policy and the wholesaler acts human-like the cost increase to 2865. Something similar happens when all players act human-like, but the wholesaler is applying the optimal base-stock policy (3664). If at least one player does not act according to the base-stock policy, the cost is no longer optimal. In fact, we observe the opposite effect. If not applied by all players, it leads to some of the highest costs observed in our experiments. We would like to draw special attention to the last row because it indicates the cost reduction if the intelligent algorithm replaces a human-like wholesaler. In the case of the co-players following the base stock policy, it leads to a cost reduction of 85,5%. If the co-players act human-like the cost decreases by 71%. Hence if we assume that not all players act according to the base stock policy, then implementing the REINFORCE Agent leads to a significant cost reduction, when the end-customer demand is classical.

Under uniformly distributed end-customer demand, the problem is more complicated because we introduce randomness. Table 2 shows the performance of our agents under the uniformly distributed demand. The system with all players acting according to the base stock policy does not get into the Nash equilibrium. It is a stochastic value now. Remember that we inserted the knowledge about the demand distribution into the random policy. We draw demand and actions from the same distribution, which leads to a random policy that already seems quite good. If we replace a human-like wholesaler by our intelligent agent, we can reduce cost in the REINFORCE-base stock (16,5%) and REINFORCE-sterman (17,7%) cases. If we have random demand and randomly acting co-players, then the REINFORCE wholesaler does not learn a strategy to reduce the cost further. In fact, the cost slightly increases by 1,6%. Compared to the performance of a human-like wholesaler, there is a reasonable cost-reduction overall. Nevertheless, when we replace the wholesaler by a random-policy instead of the RL-algorithm, we get almost equally good results. One might question the investment in an intelligent RL-agent when just randomly drawing actions reduces cost comparably well. However, for the random-policy, we assumed to know the underlying demand distribution. The REINFORCE agent, on the other hand, learns to react to the demand distribution on its own.

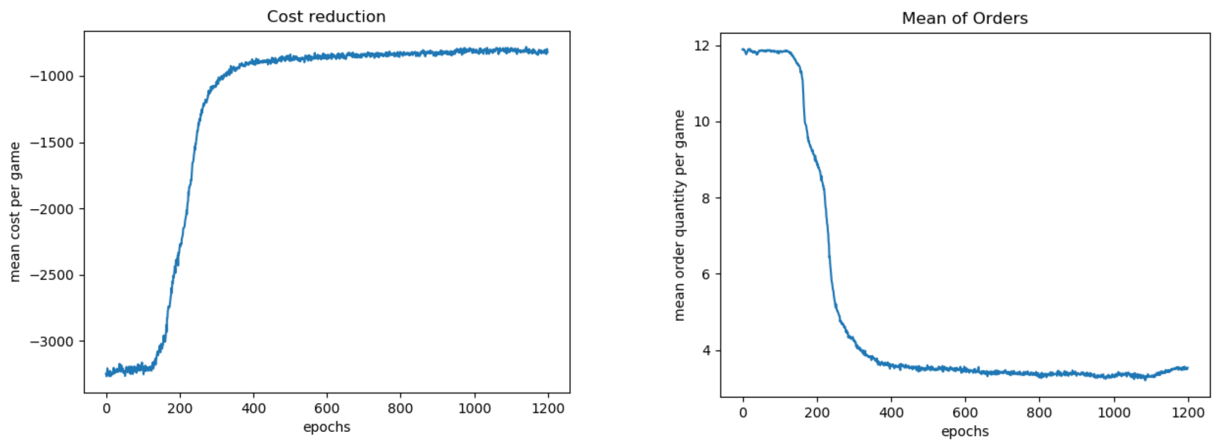


Figure 5: Cost reduction and mean of order quantities during the training process

When we have a closer look at what the RL-agents learn, we notice that the agent approximately orders the mean of the demand distribution. In the U[0,8] case, this is equal to 4 (Fahrmeir et al., 2016). We see how the cost decreases while the agent approximates the mean of the demand distribution within its orders (see figure 6).

6.2 Continuous policies

The continuous version of the BDG allows the placement of real-valued orders. The action spaces basically remain the same, but orders can now take any value, not just natural numbers. We no longer convert the order quantities calculated by policies known from the literature to integers. This leads to slightly different benchmark values (see table 3). However, those values are similar to those in the discrete case. Deviations are a result of rounding in the discrete case. For the continuous version of the BDG, we applied the REINFORCE algorithm with gaussian policy and the DDPG algorithm.

6.2.1 Reinforce with Gaussian policy

The continuous REINFORCE algorithm is applied together with a gaussian policy to handle continuity. The sigma that is delivered by the neural network is activated with a Softplus activation function. This activation function ensures that the sigma is positive. We observe some instabilities during the training process, which seems to have numerical reasons. The estimated variance gets so small that some of the log-probabilities reach infinity. Therefore, we force the estimated sigma to a minimum value of 0.001 during the calculation of the log-probabilities. We further apply gradient clipping and normalise the rewards to an [0,1] interval. We also slightly change the algorithm in a way that when we created a batch of experience, we take five consecutive update steps in a row. Those modifications stabilised learning. However, as we can see in Table 3, the performance of the continuous REINFORCE (Rf-cont) is not as good as the performance of the discrete REINFORCE, although we trained the continuous version for 1000-5000 epochs. The varying duration of training is a result of a much slower training process and later convergence. Due to computational limitations, we had to interrupt the training process there. However, under the classical demand distribution, the continuous REINFORCE algorithm reduces the cost by 74,6% when co-players act upon the base-stock policy, and the intelligent algorithm replaces a human-like wholesaler (%Gap RF-Sterm.). The cost reduction is 69,2% when playing with human-like co-players. We would like to draw special attention to the performance of our algorithm with randomly acting co-players. The cost increased by 176% and 68,2% for the classical and uniform demand distribution. Although we did extend training up to 5000 epochs, we cannot observe convergence (Appendix C) and the level of performance remains below the other algorithms.

	Co-players policy					
	classical			uniform		
wholesaler policy	random	base stock	sterman	random	base stock	sterman
random	1.815,6	1.277,5	1.319,5	1.191,2	821,5	899,3
base stock	2.996,3	180,0	3.696,0	1.189,1	683,2	1.193,0
sterman	1.775,8	2.906,3	2.046,2	1.189,5	972,4	1.045,6
RF-cont	4901,2	737,6	616,2	1.984,1	780,1	763,3
%Gap RF -Sterm.	176,0%	-74,6%	-69,2%	68,2%	-19,8%	-27,0%
DDPG	1.544,9	256,9	467,4524	2001,1	774,8	1.150,9
%Gap DDPG- Sterm.	-13,0%	-91,2%	-77,2%	14,1%	-20,3%	10,1%

Table 3: Performance of algorithms with continuous action space

6.2.2 Deep Deterministic Policy Gradient

The DDPG algorithm uses a replay buffer for learning. To fill the replay buffer, we introduce a warm-up phase of 1200 games. During the warm-up phase, actions are randomly chosen, and the transitions are added to the replay buffer. For the classical demand distribution, the DDPG algorithm is trained for 5000 epochs. The DDPG algorithm decreases the cost significantly, regardless of the policy applied by its co-players (Table 3). Especially within the DDPG-(base stock) case, the DDPG agent seems to be approximating the optimal base stock cost of 180. Compared to a human-like wholesaler, the intelligent agent can reduce cost by 91.2%. If the co-players act upon a base-stock policy, the DDPG agent decreases cost by 77.2%. Under uniformly distributed demand, we observe a slight cost increase when the DDPG plays with randomly and human-like co-players (14,1%, 10,1%). When playing with base-stock co-players cost are reduced by 20,3%. Overall, it seems like the DDPG algorithm delivers a strong performance under the classical demand but has some difficulty when demand is uniformly distributed.

6.3 Benchmarks from literature and comparison of algorithms

It is reasonable to compare our results with the results obtained by Oroojlooyjadid et al. (2017) with the DQN Network. We closely followed their methodology but applied a different class of algorithms. However, to draw a fair comparison, it is essential to check whether we use the same benchmarks. Oroojlooyjadid et al. (2017) also implement the formula of Sterman (1989). Given the information in their paper, it was not possible to recreate the exact scores they are using in their web-application. Hence we based our implementation on the mathematical model and its corresponding R-implementation of Edali and Yasarcan (2014). The minor differences should be a result of the choice of some parameters and a slightly different calculation of expected demands. Whereas they use the mean of observed demands, we apply the original exponential smoothing proposed by Sterman (1989). The sterman formula includes some parameters reflecting how the human-like player over and underreacts to certain situations. By adjusting those parameters, we can directly influence the cost. A game of 36 rounds with only human-like players leads to a total cost of 2208,5 in Oroojlooyjadid et al. (2017). To get similar cost we used alpha = 0.5 and beta = 1 in the sterman formula, which leads to the total cost of 2049. However, it was not possible to achieve the exact same values, so comparing the performance concerning the sterman formula should be done with caution.

Table 4 and 5 display the performance of our three algorithms, together with the performance of the DQN algorithm implemented by Oroojlooyjadid et al. (2017). They trained their DQN algorithms for at least 60 000 episodes which significantly exceeds the 5000 episodes of our DDPG algorithm. Our REINFORCE (RF) algorithms are trained in between 1000 and 5000 epochs including the information of 139 000 – 69 5000 episodes of the game. With more training and an optimised parameter selection, our algorithms might even further improve. Table 4 shows the performance when the co-players act according to the base stock policy. Oroojlooyjadid et al. (2017) normalised the cost. Hence the absolute values do not have a meaningful interpretation.

	Classical			Uniform		
Wholesaler	*-bs	bs-bs	Gap	*-bs	bs-bs	Gap
DQN	0,47	0,34	38,24%	960,4	799,2	20,18%
RF-discrete	415,5	180	130,85%	804,2	677,7	18,66%
RF-cont.	737,6	180	309,781%	780,1	683,2	14,18%
DDPG	256,9	180	42,72%	774,8	683,2	13,40%

Table 4: Performance comparison with co-players following the base stock (bs) policy

We draw special attention to the “Gap” column. It represents the actual performance gap. The *-bs column represents the cost occurring when one of the intelligent algorithms plays the wholesaler, and the co-players follow the base stock policy. The second column represents the cost arising when we replace the wholesaler by the base stock policy. The gap column represents the difference between those two values. It can be positive or negative, reflecting the fact that the algorithms might perform better or worse than the values in the second column.

In all cases, the gap is positive indicating that our algorithms perform worse than the bs-bs policy. This is reasonable as the base stock policy is optimal and we use its cost as a lower bound. The gap implies how close we get towards this lower bound. For the classical demand distribution, we see that the DQN and the DDPG algorithm outperform the discrete REINFORCE (RF-discrete) and continuous REINFORCE (RF-cont.). The gap of 38 % and 42% indicate that the DQN and DDPG perform similarly well. The discrete REINFORCE with a gap of 130% and the continuous REINFORCE with a gap of 309% are not compatible. When we take a look at the uniform distribution, we see that the policy-gradient methods (18,66%, 14,18%, 13,4%) outperform the action-value method (DQN = 20.18%). Whereas the continuous REINFORCE algorithm delivered the weakest performance under the classical demand distribution, it shows the second-best performance under uniformly distributed demand. Overall, the policy-gradient methods seem to handle uniformly distributed demand better when playing with base stock co-players. It is also noticeable that the DDPG algorithm performs on the same level as the DQN, although it was only trained for 5000 episodes whereas the DQN was trained for 60.000 episodes.

Table 5 displays the performance when the co-players act human-like. The *-sterm column represents the performance when an intelligent agent plays the wholesaler, and the co-player follow the sterman formula. The bs-sterm column monitors the performance when a base-stock policy replaces the wholesaler. Under the classical demand distribution, we observe that the policy-gradient methods outperform the DQN algorithm. The DDPG Method performs best with a cost reduction of 87% compared to a bs-sterm setup. When the demand distribution changes, we observe the opposite scenario. The DQN algorithm outperforms the policy-gradient algorithms. The continuous REINFORCE variant is the only policy-gradient method that is compatible with the DQN in this scenario. The DDPG performs best under classical demand but is the weakest performer when demand is uniformly distributed.

	Classical			Uniform		
Wholesaler	*-sterm	bs-sterm	Gap	*-sterm	bs-sterm	Gap
DQN	2,85	8,17	-65,12%	5,9	9,53	-38,09%
Rf-discrete	594,4	3.664,5	-83,78%	871,1	1.172,3680	-25,70%
Rf-cont	616,2	3.696,0	-83,33%	763,3	1.193,0	-36,02%
DDPG	467,5	3.696,0	-87,35%	1.150,9	1.193,0	-3,53%

Table 5: Performance comparison with co-players following the sterman formula

Overall policy gradient methods can compete with the action-value based DQN approach of Oroojlooyjadid et al. (2017). Especially the DDPG agent and DQN agent perform comparably well. The simple policy gradient method REINFORCE also delivers a highly competitive performance except for the case when the co-players apply the optimal base-stock policy, and the demand is classical. Each algorithm seems to have some strengths and weaknesses depending on the particular setup comprising end-customer demand and co-player policies.

7 Conclusion

We developed a framework to test different reinforcement learning algorithms for the Beer Distribution Game (BDG). The game has a variety of features that complicate the problem. It only has partially observable states, the actors play cooperatively, and there is no information sharing during the game. Oroojlooyjadid et al. (2017) introduced a DQN algorithm addressing those complicating features. We implemented some of their mechanisms and applied them together with policy-gradient reinforcement learning algorithms. We further extended the setup to a continuous variant of the game. Hence action and state-space get real-valued. The framework allows various configurations, where every single actor within the supply chain can individually follow its own policy. Thereby we have options to choose among one of the known policies from literature: (i) the optimal base-stock policy, (ii) human-like behaviour and (iii) a random policy. We can also replace every actor with an intelligent RL-algorithm. Within the scope of this study, we limited ourselves to the case where only one actor of the supply chain is a learning agent. Nevertheless, we suggest investigating multi-agent learning based on our framework.

In our numerical experiments, the RL-algorithms replaced the wholesaler, and its co-players followed one of the three well-known policies mentioned above. Thereby the agents faced two different end-customer demand distributions. The first demand distribution is constant after some time and was defined by Sterman (1989). The second demand distribution is uniformly distributed within a certain range. We employed a discrete REINFORCE and a continuous REINFORCE variant, as well as the Deep Deterministic Policy Gradient (DDPG). All methods are capable of finding ordering policies that lead to a significant cost reduction. However, the algorithms are quite sensitive to specific settings. Considering the classical demand distribution defined by Sterman (1989) and a human-like supply chain, then replacing the wholesaler by our intelligent agents leads to a cost reduction of at least 69%. If we examine the same setting with the uniform demand distribution, only the REINFORCE algorithms lead to a cost reduction. The DDPG algorithm even slightly increases the cost. We suggest a closer investigation of the reasons why those algorithms have strengths and weaknesses under different demand distributions. Nevertheless, in comparison to the DQN algorithm proposed by Oroojlooyjadid et al. (2017), our algorithms perform comparably well. Primarily the DDPG agent reaches the same level of performance, although either one has some benefits depending on the particular setup. Due to computational limitations, the training time and parameter selection was restricted. Hence there still might be upward potential that remains unexplored.

Policy-gradient methods have a variety of features that favour exciting research questions. They, for instance, can be used to improve supply chains trading multiple items. We demonstrated that they could significantly reduce cost and deliver similar performances as action-value approaches when solving the beer distribution game. With this thesis, we created a starting point for future research regarding policy-gradient algorithms in serial supply chains.

References:

Achaim, J. (2018). SpinningUp2018, DDPG. Retrieved from: https://spinningup.openai.com/en/latest/algorithms/ddpg.html#why-these-papers. Retrieved on: 25.08.2020.

Achaim, J. (2018). SpinningUp2018: simple_pg.py. Retrieved from: https://github.com/openai/spinningup/blob/master/spinup/examples/pytorch/pg_math/1_simple_pg.py. Retrieved on: 20.06.2020.

Achaim, J. (2018). SpinningUp2018, simple policy gradient. Retrieved from: https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html#deriving-the-simplest-policy-gradient. Retrieved on: 28.05.2020.

Bernstein, D. S., Givan, R., Immerman, N., & Zilberstein, S. (2002). The complexity of decentralised control of Markov decision processes. *Mathematics of operations research*, *27*(4), 819-840.

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.

Chen, F. (1999). Decentralised supply chains subject to information delays. *Management Science*, 45(8), 1076-1090.

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

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

Eatwell, J., Milgate, M., & Newman, P. (Eds.). (1989). Game theory. *Springer*.

Edali, M., & Yasarcan, H. (2014). A mathematical model of the beer game. *Journal of Artificial Societies and Social Simulation*, *17*(4), 2.

Edali, M., & Yasarcan, H. (2014). R-Implementation: A Mathematical Model of The Beer Game. (Version 1.0.0). *CoMSES Computational Model Library*. Retrieved from: https://www.comses.net/codebases/4161/releases/1.0.0/, Retrieved on 08.07.2020.

Fahrmeir, L., Heumann, C., Künstler, R., Pigeot, I., & Tutz, G. (2016). Statistik: Der weg zur Datenanalyse. *Springer-Verlag*.

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

Graves, S. C. (1985). A multi-echelon inventory model for a repairable item with one-for-one replenishment. *Management science*, *31*(10), 1247-1256.

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.

Lee, H L., Padmanabhan, V. & Whang, S. (1997). Information distortion in a supply chain: The bullwhip effect. *Management science*. 43.4: 546-558.

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

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Martinez-Moyano, I. J., Rahn, R. J., & Spencer, R. (2005). The beer game: its history and rule changes. *Proceedings of the 23rd International Conference of the System Dynamics Society*.

Orlov, A. (2019). beer-game-env. Retrieved from https://github.com/orlov-ai/beer-game-env, Retrieved at 25.06.2020

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 Optimisation Problems. *arXiv preprint arXiv:1708.05924*.

Peters, H. (2015). Game theory: a multi-levelled approach. *Springer*.

Polyak, B. T. (1990). A new method of stochastic approximation type. *Avtomatika i Telemekhanika*, 51(7): 98–107.

Ponte, B., Fernández, I., Rosillo, R., Parreño, J., & García, N. (2016). Supply chain collaboration: A Game-theoretic approach to profit allocation. *Journal of Industrial Engineering and Management*, *9*(5), 1020-1034.

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

Strozzi, F., Bosch, J., & Zaldivar, J. M. (2007). Beer game order policy optimisation under changing customer demand. *Decision Support Systems*, *42*(4), 2153-2163.

Tabor, P. (2018). ddpg_torch.py, Retrieved from https://github.com/philtabor/Youtube-Code-Repository/blob/master/ReinforcementLearning/PolicyGradient/DDPG/lunar-lander/pytorch/ddpg_torch.py, Retrieved at 02.08.2020

Uhlenbeck, G. E., & Ornstein, L. S. (1930). On the theory of the Brownian motion. *Physical review*, *36*(5), 823.d

Wu, D. Y., & Katok, E. (2006). Learning, communication, and the bullwhip effect. *Journal of operations management*, *24*(6), 839-850.

A Appendix

n_observed rounds	5			n_observed rounds	10	
	discount				discount	
beta	1	0.95		beta	1	0.95
10	743,455	854,904		10	859,355	525,1535
100	**635,42**	1579,036		100	**514,109**	537,7645

B Appendix

BDG Parameter Selection

Cost hold	[0.5, 0.5, 0.5, 0.5]	
Cost stock	[1, 1, 1, 1]	
Lship	[2,2,2,2]	
L lead	[2,2,2,1]	
Demand_dist	{classical, uniform_0_8}	
M (n_observed_periods)	5	
N (n_turns_per_game)	36	
Discrete	{true, false}	
Action_Space	16	

Parameters for REINFORCE Training

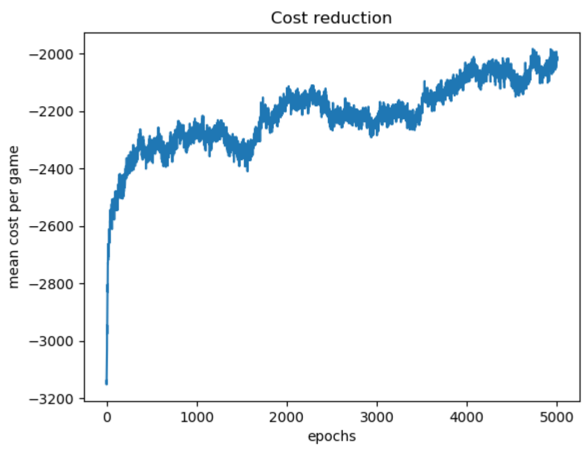
Learning_rate	0.0001	
Sizes	[20, 32,32]	Input Size & Sizes of hidden layers
n_actions	16	Size of output
Batch_size	5000	
Beta	100	Feedback (Formula 14)
Gamma	1	Reward (Formula 3)

Parameters for DDPG Training:

Alpha	0.000025	Learning rate actor network
Beta	0.00025	Learning rate critic network
Tau	0.001	
Batch_size	32	
Layer1_size	400	
Layer2_size	300	

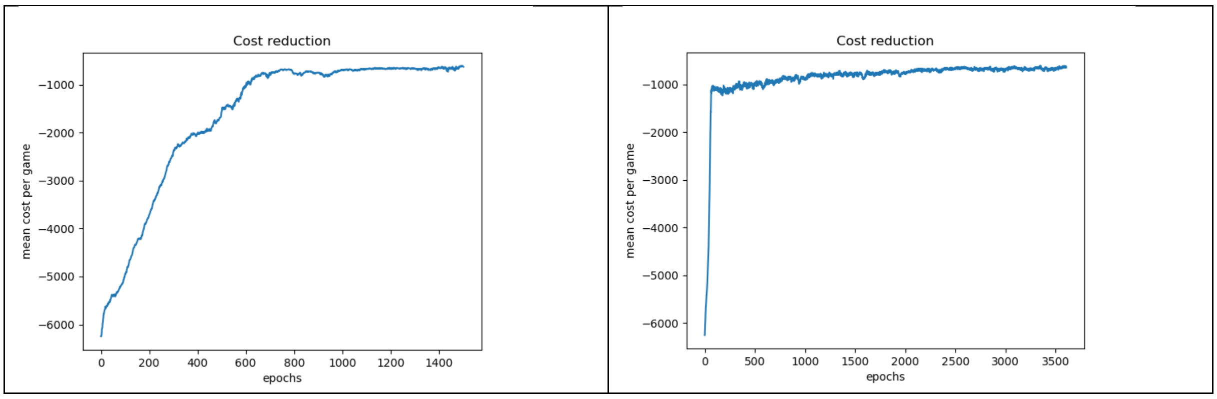
C Appendix

Training process of the continuous REINFORCE-Random setup. No convergence observed after 5000 training epochs



D Appendix

We trained the REINFORCE with gaussian policy under classical demand and co-players following the Sterman formula. On the right figure the continuous orders have been mathematically rounded, whereas on the left the continuous version of the game was played (no rounding). When evaluating both algorithms the continuous REINFORCE leads to cost of 616 and the rounded continuous REINFORCE leads to cost of 636.


Cost reduction
mean cost per game
epochs
Cost reduction
mean cost per game
epochs