Monitoring of Agents for Dynamic Pricing in different Recommerce Markets

Monitoring von Agenten zur dynamischen Bepreisung in unterschiedlichen Recommerce-Märkten

Nikkel Mollenhauer

Universitätsbachelorarbeit
zur Erlangung des akademischen Grades

Bachelor of Science
(B. Sc.)

im Studiengang IT-Systems Engineering

eingereicht am 30. Juni 2022 am Fachgebiet
Enterprise Platform and Integration Concepts
der Digital-Engineering-Fakultät der Universität Potsdam

Betreuer

Dr. Michael Perscheid
Dr. Rainer Schlosser
Johannes Huegle
Alexander Kastius

Abstract

Sustainable recommerce markets are growing faster than ever. In such markets, customers are incentiviced to re-sell their used products to businesses, which then refurbish and sell them on the secondary market. However, businesses now face the challenge of having to set three different prices for the same item: One price for the new item, one for its refurbished version and the price at which items are bought back from customers. Since these prices are heavily influenced by each other, traditional pricing methods become less effective. To solve this dynamic pricing problem, a simulation framework was built which can be used to train artificial vendors to set optimized prices using Reinforcement learning algorithms. Before employing these trained agents on real markets, their fitness must be monitored and evaluated, as prices that are too high or too low can lead to high costs for the business. This thesis introduces a number of ways that such dynamic pricing agents can be monitored. We come to the conclusion that it is best to use a wide range of tools when evaluating different aspects of an agent's performance, from running large-scale simulations to monitoring small policy changes following shifting market states.

Zusammenfassung

Nachhaltige Recommerce-Märkte befinden sich in stetigem Wachstum. Diese Märkte bieten Kunden Anreize, ihre gebrauchten Produkte an Unternehmen zurückzuverkaufen. Diese generalüberholen das angekaufte Produkt und stellen es anschließend auf dem Sekundärmarkt erneut zum Verkauf. Dies stellt Unternehmen jedoch vor die neuartige Herausforderung, dasselbe Produkt mehrfach bepreisen zu müssen: Preise sowohl für die neue und generalüberholte Version sowie ein Ankaufpreis für gebrauchte Ware müssen gesetzt werden. Da diese Preise einander beeinflussen, greifen traditionelle Methoden der Preissetzung schlechter. Zur Lösung dieses dynamischen Bepreisungsproblems wurde eine Simulationsplattform gebaut, auf der mithilfe von Reinforcement-Learning-Algorithmen maschinelle Verkäufer für den Einsatz in realen Märkten trainiert werden können. Bevor dies jedoch geschehen kann, müssen die trainierten Modelle bezüglich ihrer Eignung überprüft und bewertet werden, da zu hoch oder niedrig angesetzte Preise zu hohen Verlusten aufseiten des Unternehmens führen kann. Diese Arbeit führt Tools ein, die für ein Monitoring solcher Modelle verwendet werden können. Wir stellen fest, dass die Nutzung möglichst diversifizierter Methoden, von der Simulation großangelegter Märkte bis zum Monitoring kleinster Verhaltensänderungen aufgrund geänderter Marktzustände, die besten Ergebnisse bei der Bewertung einer Bepreisungsmethode liefert.

Contents

1 Introduction

This thesis builds upon the bachelor's project 'Online Marketplace Simulation: A Testbed for Self-Learning Agents' of the Enterprise Platform and Integration Concepts research group at the Hasso-Plattner-Institute. Therefore, the project will be referenced and all examples and experiments will be conducted using its framework.

1.1 Motivation

Nowadays, shoppers and retailers alike are getting more environmentally conscious. A study conducted in 2020 found that over two-thirds of shoppers planned on buying sustainable clothing in the future, and over half already did buy sustainable clothing regularly [KPM20]. At the same time another study reveals that retailers favour online channels over offline channels when selling used goods, with 78% preferring the former and only 6% the latter [Ins20]. This demand is leading more and more companies, especially those selling their products through e-commerce channels, to adopt more sustainable strategies and enter the *Circular Economy*, a concept we will explain in Section 1.2. In the context of e-commerce, Circular Economy markets are also referred to as *recommerce* markets, a phrase coined in 2005 [Arc05].

The goal of the bachelor's project this thesis is based on was to simulate such *recommerce* markets, to allow for risk-free and efficient learning of market characteristics for different pricing agents. Aside from classically rule-based pricing methods, the project also focussed on training machine learning models using *Reinforcement learning* (RL) algorithms on the simulated marketplace, see Section 1.3. The focus of this thesis lies on ways that such dynamic pricing agents can be monitored and evaluated. In Chapter 2 we will first explore the difficulties faced in dynamic pricing, as well as current and novel approaches to monitoring and evaluating Reinforcement learning agents in different scenarios. This will be followed by an overview of the specific features of the *recommerce* market that we implemented in Chapter 3. Chapter 4 contains detailed explanations of the different tools we built and used to monitor and evaluate our different agents. These tools will be put into context in Chapter 5, where the simulation workflow of the framework will be introduced. Finally, in Chapter 6, we will train an RL agent using our simulation framework and take a look at how the different monitoring tools can help in evaluating its performance.

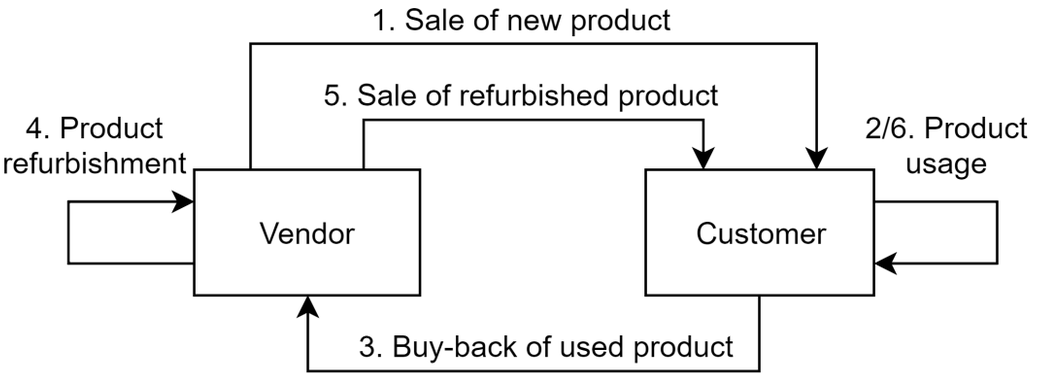


Figure 1.1: The product lifecycle in a Circular Economy (with rebuy prices). In a Linear Economy the product lifecycle ends with step 2. Step 6 may also reconnect with Step 3 to start a new cycle.

1.2 The Circular Economy model

The main goal of the aforementioned bachelor's project was to simulate an online marketplace that reconstructs a realistic Circular Economy market environment. A market is most commonly referred to as being a *Circular Economy* if it includes the three activities of reduce, reuse and recycle [KRH17]. This means that while in a classical Linear Economy market each product is being sold once at its *new price* and after use being thrown away, in a Circular Economy a focus is put on recycling and thereby waste reduction. In our project, we first started by modelling the simpler Linear Economy, upon which we then built the more complex Circular Economy markets. This was done by adding two additional price channels, *refurbished price* and *rebuy price*, to the pre-existing *new price* of a product. Please also refer to Figure 1.1 for an overview of the product lifecycle in a Circular Economy with rebuy prices.

The *rebuy price* is defined as the price a vendor is willing to pay a customer to buy back a used product, while the *refurbished price* is defined as the price the vendor sets for products they previously bought back and now want to sell alongside new products (whose price is defined by the *new price*). In Section 3.1 we will explain the different market scenarios we modelled in more detail, together with how we transferred different aspects of these scenarios from the real market to our simulation framework. Please also refer to Figure 3.1 for an overview of the different market components and how they interact in our framework.

From now on, when referring to a *recommerce market*, we are referencing a Circular Economy marketplace with rebuy prices.

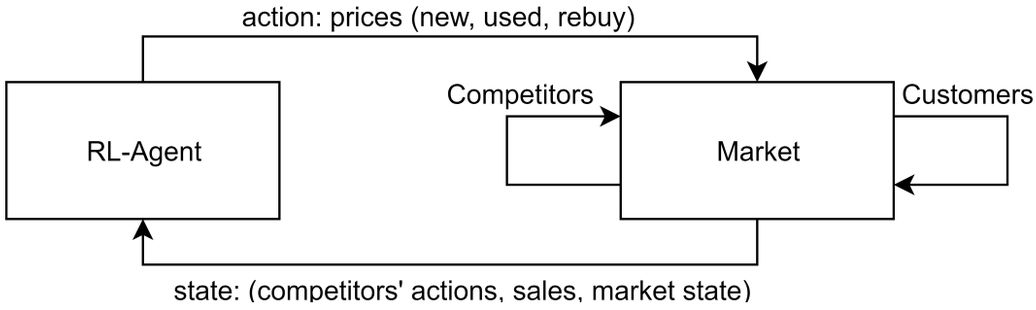


Figure 1.2: The standard Reinforcement learning model in the context of our market simulation.

1.3 Reinforcement learning

After the initial market was modelled, the goal was to train agents using different Reinforcement learning algorithms to dynamically set prices on this marketplace, both in monopolistic scenarios and in competition with rule-based vendors which set prices following a strict set of pre-defined rules. These rules can range from simply undercutting the lowest competitor's price to more advanced techniques such as smart inventory management and reliance on previous sales data. An overview of the different types of vendors, both rule-based and using RL can be found in Section 3.3. Furthermore, functionality was added that allows for different RL algorithms to be trained against each other on the same marketplace, as well as functionality for so-called *self-play*, where an agent plays against itself, or more precisely, against its own policy, see [Gro22] for more information.

Reinforcement learning agents are trained through a process of trial-and-error. They interact with the market through an observable state and an action which influences the following state. Figure 1.2 illustrates the RL model in the context of our market. The goal of the agent is to maximise its *reinforcement signal*, which in the case of our simulation framework is the profit the agent made during the last training episode, since we want to train agents to maximise profits on real markets. An episode consists of a fixed, but pre-configurable number of time steps, where in each step each vendor (agent) sets their prices and customers make purchasing decisions. In this sense, a simulated episode could be imagined to be a day in the real world, with vendors updating their prices multiple times per day, whenever a new time step begins. By observing which prices lead to which profits (= the reinforcement signal), the Reinforcement learning agents get more effective in their pricing decisions over the course of training, which in most cases spans thousands of episodes.

2 Related Work

This chapter introduces the history and importance of dynamic pricing methods and their connection to the demand for simulated marketplaces. Additionally, traditional and novel approaches to monitoring and evaluating Reinforcement learning agents will be presented.

2.1 Dynamic Pricing

The topic of dynamic pricing techniques is well explored, with the earliest mathematical models having been developed over a century ago [den15]. With the emergence of e-commerce and the ability to cheaply and quickly change prices, as well as increased freedom of information, including being able to know competitors' prices in real time, the topic has gained importance even further. However, research concerning dynamic pricing in e-commerce, especially in highly competitive markets, has not grown at the same pace as the markets themselves [GB22]. On the other hand, the role that autonomous agents will play in this new market environment, in the form of vendors and customers alike, has for a long time been a topic of discussion and speculation [KHG00]. Dynamic pricing methods come in many different shapes and forms, from simple greedy algorithms over customer-choice methodology to machine-learning models [Gee+19]. Due to this high number of options to choose from, and to enable researchers to evaluate their algorithms' performance, a unified platform for comparison is needed. The real market is not really an option for this, as it is nearly impossible to create equal opportunities for each pricing agent to be able to reliably compare them to each other. This is one of the major reasons for simulating the real market - development, testing and comparison of new pricing methods.

2.2 Visualisation - State-of-the-art

When training RL agents, it is almost a requirement to be able to visualise data collected during training, to allow for an analysis of the algorithm's performance.

For the past years, going back as far as 2018, one of the most used programming frameworks overall and the most used for machine learning was TensorFlow [Ove22]. Aside from its API for model building, TensorFlow also provides a front-end visualisation toolkit called TensorBoard, which can be used independent of other TensorFlow tools. TensorBoard provides an API for tracking and visualising important metrics such as loss and accuracy of a trained model, and allows developers to easily integrate their own metrics as well. During an experiment, data can be visualised live during the training process, allowing developers to quickly gain insights into the performance of the algorithms. In our market simulation framework, we use the TensorBoard in conjunction with our own tools for data visualisation, see Chapter 4.

2.3 Visualisation - Novel approaches

2.3.1 Model visualisation

Aside from visualising results of a training run, developers may also want to visualise the model itself. Tools such as the *Graph Visualizer* [Won+18] are a step in the right direction, but the authors found that while developers are satisfied with the visualisation tool itself, they would prefer to be able to edit and influence the graph model directly. The *SimTF* tool developed by Leung et al. [CLH18] attempts to solve this problem. The same as the *Graph Visualizer*, the *SimTF* tool is based on TensorFlow. The authors describe it as a *library for neural network model building*. *SimTF* acts as a middleman between the visually constructed neural network graph model and the TensorFlow API, allowing developers to modify the visualised model to influence the training network.

In our simulation framework, the used neural network model itself is quite static, as users can only choose from a pre-set selection of network sizes, and all changes to hyperparameters must be done through configuration files ahead of a training run.

2.3.2 Evaluation

On the other side of the visualisation tools we have those which provide insights into currently running or completed training runs, such as the previously mentioned *TensorBoard*. Besides simply visualising different collected metrics, the goal of most of these tools is to allow the developers to get a sense of the trained algorithm's performance, and to evaluate and compare it against previously trained agents using the same algorithm or other algorithm's completely.

When developing new algorithms, the most common barrier to proper comparison with current state-of-the-art algorithms is a lack in reproducibility of results. Benchmark environments, such as those provided by OpenAI Gym [Bro+16], lower this barrier by providing unified interfaces against which many algorithms have already been tested. However, the effects of extrinsic factors (such as hyperparameters) and intrinsic factors (such as random seeds for environment initialisation) make proper, reliable reproducibility a challenge, something for which current evaluation practices do not account for [Hen+17]. Jordan et al. [Jor+20] propose a new, *comprehensive evaluation methodology for reinforcement learning algorithms that produces reliable measurements of performance*. The authors describe the goal of their new evaluation technique as not trying to find methods (algorithms) that maximise performance with optimal hyperparameters, but rather to find those that do not require extensive hyperparameter tuning and can therefore easily be applied to new problems. This leads to a preference for algorithms that are not aimed at being the best at problems which are already solved (which applies to the aforementioned benchmark environments), but instead those which are most likely to succeed in new, unexplored environments.

In our framework, we attempt to provide users with visualisations for as many different metrics as possible, to enable them to properly gauge the fitness of the currently tested dynamic pricing method.

3 Simulating the Marketplace

This chapter will introduce the various components that make up a recommerce market and how they were implemented in our simulation framework. We will take a brief look at different market scenarios as well as how our customers make decisions. The focus of this chapter will however lie on the vendors, the rule-based as well as the ones trained using Reinforcement learning algorithms. For further information on the overall framework structure refer to [Dre22], for detailed insights into our market processes, see [Bes22].

3.1 Market scenarios

In our framework, we implemented a number of 'market blueprints' for classic Linear Economy markets as well as for Circular Economy markets both with and without rebuy prices. There are marketplaces available for each combination of the following two features:

1. Marketplace type: Linear Economy, Circular Economy, Circular Economy with rebuy prices

2. Market environment: Monopoly, Duopoly, Oligopoly

The marketplace type defines the number of prices the vendor has to set. In a Linear Economy vendors only set prices for new items, in a Circular Economy prices for refurbished items need to be set as well. Without rebuy prices, customers simply return products to vendors 'for free'. In order to simulate a proper *recommerce* market, users can choose a Circular Economy with rebuy prices.

The market environment defines the number of competing vendors in the simulation: Monopolistic and competitive markets are available, with the Duopoly simply being a particular version of an Oligopoly with only two competing vendors. Depending on the chosen market environment one, two, or any number of vendors can be chosen to be used in an experiment. Figure 3.1 shows a reduced overview of the classes in our framework which concern themselves with the market simulation and how they interact with each other during the simulation.

In the most common use case of our framework, the training of RL agents, only the agent (or multiple agents in the case of training multiple RL agents against each other) that is to be trained, needs to be configured by the user, as each market environment is equipped with a pre-defined set of competitors that will play against the agent defined by the user. To allow for more control over the simulation, users are however also able to change these competitors to use any combination of vendors they want - as long as they are a valid fit for the marketplace type and the number of chosen competitors matches the market environment. How to configure an experiment will be explained in Chapter 5.

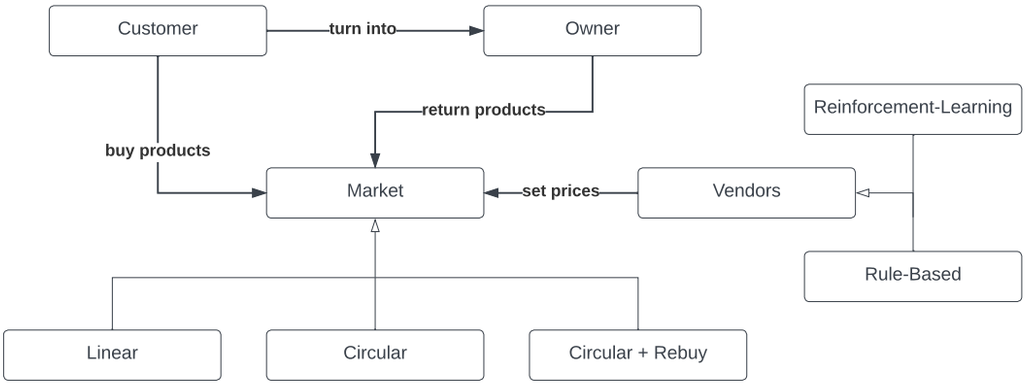


Figure 3.1: Interactions between classes concerning the market simulation.

3.2 Customers

Customers are at the centre of every type of marketplace, which makes them an integral part of our market simulation. However, since each customer in the real market is an individual with different backgrounds and makes purchase decisions based on personal preferences, modelling a realistic depiction of real-world customers proves to be very difficult and time-consuming. For this reason we decided to keep our initial implementation of the customers as simple as possible, taking into account future extension and scalability concerns.

Most customers' behaviour can be classified into one of a (non-exhaustive) number of categories, such as those proposed in [EIT13], see Table A.1. As we are focussing on dynamic pricing and our vendors can only change/influence the prices of their products, we decided on primarily building customers that value price over any other features a product may have, thereby incentivising our vendors to make the most of their pricing power. This behaviour coincides with the proposed shopping style of the *Price Conscious* or *Value for money* consumer.

To make Linear markets a little more complex and add another dimension than just the *new price* of a product for vendors to consider, random quality values are assigned to each vendor's products at the start of an episode. Customers in this economy model were modelled to take this quality value into account when making their purchasing decisions, further reinforcing the shopping style mentioned above. As Circular Economy markets are inherently more complex than their linear counterparts (through the addition of two new price channels, which influence each other through their effect on customer behaviour), it was decided to remove the additional layer of quality values from these markets.

Within each simulated time step, after the various vendors have set their prices, purchasing decisions are made by the customers. To save on computational time, probability distributions are used to determine what part of the total number of customers will decide to take which action, instead of simulating each customer individually. See Definition 3.1 and Definition 3.2 for the way these distributions are calculated for an exemplary recommerce marketplace, for further information also see [Bes22].

► **Definition 3.1.** Let $P_{i,new}$ be the price of the new and $P_{i,ref}$ the price of the refurbished product set by vendor i, with $0 \leq i < n$ and $n \in \mathbb{N}$ the number of vendors in the market. Prices will always be within the range $[0, 10]$. We now define $r_{new}(P)$ as the function determining the customers' *preference ratio* regarding a new price P set by any vendor:

$$r_{new}(P) := \frac{10}{P} - e^{P-8} \tag{3.1}$$

and similarly, $r_{ref}(P)$ as the preference ratio regarding the refurbished price:

$$r_{ref}(P) := \frac{5.5}{P} - e^{P-5}. \tag{3.2}$$

Additionally, there is a static preference r_{not} that any customer will opt to buy no product:

$$r_{not} = 1. \tag{3.3}$$

◄

Following the numerators of the two fractions, customers adjust their determined cost-benefit ratios to value a refurbished product to be 55% the quality of that of a new product. This adjustment is necessary as there is no inherent quality value for products in our simulated recommerce markets. Additionally, the constants of 8 and 5 respectively in the exponent of the substracted exponential function act as price thresholds above which it becomes exponentially more unlikely that a customer will choose the respective product, as the value of the preference ratio will decrease.

Definition 3.2 explains how we use the *softmax* function to normalise our values, after which each preference ratio will be within the interval $(0, 1)$, with all ratios adding up to 1, as is required to use them as a probability distribution. We use *softmax*, as it is able to deal with the possibly negative results of the two r-functions.

► **Definition 3.2.** We keep the definitions from Definition 3.1. Now, let S be the sum of the exponentials of the preference ratios for all n vendors and the preference to buy nothing:

$$S := e^{r_{not}} + \sum_{i=0}^{n} \left(e^{r_{new}(P_{i,new})} + e^{r_{ref}(P_{i,ref})} \right). \tag{3.4}$$

We can now normalise the preference ratios and determine the concrete probabilities of customers choosing to buy the new, refurbished or no product by using the *softmax* function on our preference ratios. Let $\pi_{i,j}$ be the probability that a customer chooses to buy product j from vendor i, with $0 \leq i < n$ and $j \in \{new, ref\}$:

$$\pi_{i,j} := \frac{e^{r_j(P_{i,j})}}{S} \tag{3.5}$$

and let π_{not} be the probability of a customer buying no product:

$$\pi_{not} := \frac{e^{r_n}}{S}. \tag{3.6}$$

All of these probabilities are collected in the set Π. Following the definition of *softmax*, all probabilities in Π sum up to 1, as required. ◄

Following this conversion of the preference ratios, the market uses a standard *multinomial* distribution to draw m samples out of the pool of probabilities Π, with $m \in \mathbb{N}$ being the number of customers making a purchasing decision in this step of the simulation. The result of this sampling decides how many customers buy which product. See Section A.2 for an exemplary calculation of such distributions.

As mentioned at the start of this section, the current decision-making process of customers in our simulation is still quite basic. Section 7.1 introduces a number of parameters and circumstances that can be used to make customer behaviour more realistic. Through the modular approach when building the framework, any customer behaviour implemented in the future can easily be added to the pool of available options, as long as it manifests in the form of a probability distribution. The total number of customers can also be split between different distributions when drawing from the *multinomial* distribution (or any other distribution if so desired).

3.2.1 Owners

Once a customer has decided to buy a product from any of the available vendors, they turn into an *owner*. In the next step of the simulation, they are offered the option of selling their now used product back to one of the vendors. If they decide to do so, the vendor pays them the advertised *rebuy price* and adds the used product to their inventory, from where it can then be sold as a refurbished product in the following step.

In our simulation, all owners are memoryless, meaning that they do not remember the original price they paid for the product. Additionally, each vendor in the market is obligated to buy back any product, independent of the original vendor. In each step every owner has the option to either keep the product, discard it (meaning it is removed from the market and not sold back to a vendor), or sell it to any one of the vendors in the market. Similar to the way we compute customer purchasing decisions, the decisions owners take are also represented through probability distributions. Within each time step, a constant percentage of all owners, currently defined to be 5%, will either return or discard (throw away) their product, with the rest of the owners keeping their product for at least one more time step. When deciding what to do, owners act according to the following preference ratios:

► **Definition 3.3.** We keep the variables defined in Definition 3.1. Additionally, let $P_{i,back}$ be the price at which vendor i is willing to buy back an owner's product (the *rebuy price*). $P_{i,back}$ will be within the range of $[0, 10]$. For each vendor i, we define $P_{i,min}$ as the vendor's purchase option with the lowest price:

$$P_{i,min} := \min(P_{i,new}, P_{i,ref}). \tag{3.7}$$

We define $r_{back}(P)$ as the function determining an owner's *preference ratio* to sell their product to vendor i, $0 \leq i < n$ regarding the rebuy price:

$$r_{back}(P) := 2 \cdot e^{\frac{P_{i,back} - P_{i,min}}{P_{i,min}}}. \tag{3.8}$$

Additionally, the owner's *discard preference* r_{disc} is updated for each vendor as follows:

$$r_{disc} := \min\left(r_{disc}, \frac{2}{P_{i,back} + 1}\right). \tag{3.9}$$

Meaning the owner is more likely to discard their product if rebuy prices are low across the board. ◀

Equal to the way our customers work, we again normalise these preference ratios using the *softmax* function defined in Definition 3.2 (substituting r_{not} with r_{disc} and the ratios for new and refurbished products with r_{back}) and subsequently draw samples from a *multinomial* distribution.

3.3 Vendors

Vendors are the main focus of our market simulation. While our framework will not be able to reproduce all types of pricing models used in the real market, we strive to model as many different models as possible (and feasible in the scope of the project). Due to the modular nature of the framework, it is possible for users to easily create and add their own pricing strategies in the form of a vendor that can play on a market. This allows users to not only add other RL algorithms, but also to define new rule-based strategies. For this reason, we also do not restrict the usage of our monitoring tools to just RL agents, but allow users to monitor type of vendor, independent of the underlying way in which it computes prices.

We will use four types of dynamic pricing models, as defined in 'Dynamic pricing models for electronic business' [Nar+05], to describe the different models we implemented in our framework. These categories are not mutually exclusive, and many agents belong to more than one category.

3.3.1 Inventory-based models

These are pricing models which are based on inventory levels and similar parameters, such as the number of items in circulation (items which are currently in use by customers). In our framework, almost all rule-based agents consider their inventory levels when deciding which prices to set. The only exception to this rule are the simplest of our agents, the *FixedPriceAgents*, which will always set the same prices, no matter the current market state and competitor actions. The prices these agents will set are pre-determined through the user's configuration of the experiment and will not change over the course of the market simulation.

Inventory-based models are comparatively easy to implement, as they only depend on data immediately available to the vendor. This has the advantage that rule-based agents which fall into this category are relatively simple to create and modify. Examples of *Inventory-based agents* in our framework can be found in the *RuleBasedCERebuyAgent*, one of the first rule-based agents we created. This agent does not take pricing decisions of its competitors in the market into account, but simply acts according to its own storage costs, always trying to keep a balance between having enough (refurbished) products to sell back to customers and keeping storage costs low. It does this by checking into which of four possible ranges the current number of stored products falls. The more products the agent already has in storage, the lower it sets the rebuy price (to 'prevent' customers from returning

more products) and the lower it sets the price for refurbished products, to empty the storage more quickly and prevent high storage costs. While its performance is not necessarily bad, it is still one of the weakest competitors currently available in the framework. For the full implementation logic of the agent's policy, see Figure A.1. For a comparison of the *RuleBasedCERebuyAgent* with a more sophisticated *Data-driven model* see Figure 6.8.

3.3.2 Data-driven models

Data-driven models take dynamic pricing decision-making one level further. They utilise their knowledge of the market, such as customer preferences, past sales data or competitor prices to derive optimal pricing decisions. Aside from the aforementioned *FixedPriceAgents* and the basic *RuleBasedCERebuyAgent*, all of our other rule-based agents fall into this category. One of the most prominent examples of a *Data-driven model* is the *RuleBasedCERebuyAgentCompetitive*, an agent whose basic goal is to always try and undercut competitor prices, while also keeping track of the current amount of items in storage. In its policy, this agent first always undercuts the new price of all other competitors. Then, similar to the *RuleBasedCERebuyAgent*, the agent looks at the current amount of stored products and depending on the range, sets lower prices for refurbished items and higher rebuy prices, while always trying to give customers a better deal than any of the competitors. Again, the full implementation logic of this vendor's policy is available in the Appendix, under Figure A.3.

Notably, all of our *Data-driven models* are also *Inventory-based* to a certain extent, as handling storage plays a big part in a Circular Economy market setting where used products need to be bought back from customers and subsequently undergo refurbishment while in inventory of the company. *Data-driven models* have proven to be the most competent rule-based agents in our recommerce market scenario, in particular the above described *RuleBasedCERebuyAgentCompetitive*, which is able to outperform most other rule-based agents, both *Inventory-based* and *Data-driven models* (please refer to e.g. Figure A.11 (a)).

3.3.3 Game theory models

Game theory concerns itself with the study of models for conflict and cooperation between rational, intelligent entities [Mye97]. It is therefore often applicable in situations where competing individuals, acting rationally and selfishly, interact with each other. In our framework, most competitors in the market are influenced in their decision-making processes by the actions of other participants of the scenario. This is especially true for the Reinforcement learning agents, which base their policy on the received market states, which include their competitor's actions. While none of our rule-based agents have been specifically designed to act according to game theoretic strategies, due to the fact that almost all of them consider their competitors prices in their pricing decision and due to the nature of RL trying to maximise their own profits without regard to their competitors performance, behaviour according to Game theory can sometimes be observed. This can be observed especially well in the beginning of training sessions, where rule-based competitors often struggle to achieve high profits while the RL agent is making great losses, but are then able to increase their profits as the trained agent starts to

make better decisions as well. Many examples of such behaviour can be found in the diagrams shown in Chapter 6, such as in Figure 6.1.

During training RL agents observe the market state, which includes prices and sales data not only of themselves but also the other vendors in the market. Using this data, the algorithms try to predict how their prices will influence customer behaviour as well as the competing vendors. In some cases, depending on the concrete behaviour of the competitors, vendors may cooperate in driving prices higher together. In other cases the agents may act seemingly irrationally, lowering prices in order to force their opponents to lower theirs as well. An example of such behaviour can be seen in the development of prices for new items set by the two competing vendors in Figure 6.3 (d).

3.3.4 Machine learning models

All of our Reinforcement learning agents fall into this category. As they are not the focus of this thesis, we will not go into detailed explanations of the various models used. For a comparison of the performance of different Reinforcement learning algorithms in the context of our simulation framework, please refer to [Gro22]. The following will give a short overview over the different algorithms used in our framework.

There are two 'origins' of the algorithms in our framework. In the earlier phases, Q-Learning and Actor-Critic algorithms were custom implemented by us. Later on we used the Stable-Baselines library to incorporate a greater number of pre-implemented algorithms into our framework. While these are not as easily configurable, they can quickly be used without much additional work.

Q-Learning: *Q-Learning* agents were the first RL agents we introduced in our framework, as its algorithm is one of the easier ones to implement [KLM96]. The *Q-Learning* algorithm used in our framework is implemented using the *PyTorch* framework (see [Pas+19]). However, the drawback of using *Q-Learning* in our framework is that it can only be applied to discrete action and state spaces. This means that when using *Q-Learning* only 'whole' prices can be set by the vendors and any decimal places must be omitted. This of course limits the framework in its realism, as the fine-tuning of prices using decimal places can be critical in influencing customer decisions. In the initial exploration-phase of our simulation framework this was not a problem, as relatively small action spaces were used, adapted to this limitation. But by now, our simulation framework also supports continuous action and state spaces, which allows more complex algorithms such as those introduced in the sections below to function. While approaches for *Q-Learning* algorithms that can work with continuous action and state spaces have been presented in the past [GWZ99], [MPD02], we have chosen not to implement such an algorithm in our framework, but rather explore approaches other than *Q-Learning*, such as *Actor-Critic* algorithms, introduced below.

Actor-Critic: *Actor-Critic* algorithms are more complex than Q-Learning algorithms, and have therefore been implemented later in the process. They are structured different then Q-Learning algorithms in the way that they are 'split' into two parts: The *actor* is responsible for selecting actions, while the *critic* is responsible for critiquing the actions taken by the *actor*, thereby improving its performance [SB18]. Similar to the *Q-Learning* agents, the *Actor-Critic* algorithms have also been implemented by us. In total, one discrete and two continuous *Actor-Critic* agents can be used, in addition to those provided through *Stable-Baselines*.

Stable-Baselines: Stable-Baselines provides a number of open-source implementations for various RL algorithms. In our framework we use the latest version of these algorithms, through *Stable-Baselines3* [Raf+21]. The advantage when using algorithms provided through Stable-Baselines lies in the fact that they need close to no custom implementation from our end, we can instead interact with them through very simple interfaces. This cuts down on the amount of time and effort that needs to be spent developing, implementing and maintaining these powerful algorithms, and allows us to introduce a higher number of algorithms than would otherwise be possible.

Currently, five different *Stable-Baselines3* algorithms are used in our simulation framework, see the *Stable-Baselines3* documentation [Bas22] and the referenced papers for more information about the different algorithms:

- *A2C*: Advantage-Actor-Critic [Mni+16]
- *DDPG*: Deep deterministic policy gradient [Lil+15]
- *PPO*: Proximal Policy Optimization [Sch+17]
- *SAC*: Soft Actor-Critic [Haa+18]
- *TD3*: Twin-delayed deep deterministic policy gradient [FHM18]

We will be using some of these algorithms in our experiments and briefly compare them with our rule-based approaches using our various monitoring tools which are introduced in the next chapter, Chapter 4.

4 Approaches to Monitoring Agents

In this chapter we will take a look at the different approaches we took to monitoring agents in our framework, explaining the reasons why we chose to implement specific features and how they help us in determining an agent's strengths and weaknesses.

4.1 When to monitor what

Our *workflow* (which will be explained in more detail in Chapter 5) can generally be split into two parts when it comes to monitoring and evaluating agents: *during* and *after* training. When talking about the *workflow* we refer to the process of configuring and starting a training session, where a Reinforcement learning agent is being trained on a specific marketplace against competitors. The *workflow* also includes the subsequent collection of data used to evaluate an agent's performance. We are also introducing the term of the *complete agent* in this section, which will be used to refer to both RL agents that have completed a training run and rule-based agents, which do not need training.

As mentioned above, we split our monitoring tools into the following two categories:

1. **During training** (Section 4.2): Having data available as soon as possible without having to wait for a long training session to end is crucial to an efficient workflow. Our framework enables us to collect and visualise data while a training session is still running. This allows users to always be well-informed about the currently running experiments. In some cases, when an agent's performance is severely lacking, users may want to stop a training session before it has finished, which is enabled through these monitoring tools. We also include the *Live-monitoring* tool in this category, which runs directly after a training session has finished, see Section 4.2.2.

2. **On complete agents** (Section 4.3): After a training session has finished we have a complete and final set of data available for an agent, this enables us to perform more thorough and reliable tests. These can include simulating runs of a marketplace to gather data on the agent's performance in different scenarios and against different competitors, or running a static analysis of the agent's policy in different market states. The tools available for trained agents are in the same way also usable on rule-based agents.

In the following sections, we will take a look at the tools our framework provides for monitoring agents, distinguishing between the two general types of monitoring mentioned above. The goal of these sections is to give a short overview of each tool, how and why they were implemented and what value they offer to the framework as a whole. We will also discuss some features that are not yet available, explaining how they could benefit the entire workflow and enrich the overall experience.

4.2 Monitoring during training

When talking about monitoring agents during a training session, we are always referring to RL agents, as rule-based agents always perform the same and cannot be trained. But even though they cannot be trained, our monitoring tools listed in the next section, Section 4.3, can still be applied to rule-based agents as well, as users may want to compare different rule-based strategies against each other or measure the strategy's performance on a market before training an RL agent against it.

Monitoring agents while they are still being trained enables us to be more closely connected to the training process. Ultimately, the goal of such monitoring tools is to be able to predict the estimated 'quality' of the final trained agent as reliably as possible while the training is still going.

4.2.1 TensorBoard

The *TensorBoard* is an external tool from the RL library *TensorFlow* [Aba+16]. With just a few lines of code a training session can be connected to a TensorBoard instance. We are then able to pass any number of parameters and metrics we deem interesting or useful to the TensorBoard, which then offers visualisations for each of them, updating live as the training progresses. In addition to metrics specific to our market simulation, which can be found in Table A.3, the TensorBoard also visualises a number of specifically training-related data points, such as the number of episodes simulated per second. To access these web-based visualisations, a local server needs to be started. The diagrams created using the TensorBoard are an immensely useful tool for quickly and easily recording data and offering a first rough comparison of competitors in the market. Aside from simple visualisations, TensorBoard also offers many plugins and even enables users to write their own [Ten20]. Plugins such as the *What-If Tool* [Wex+20], [PAI22], which allows users to feed trained models with hypothetical situations to study their behaviour, can have a great influence on the way users interact with the TensorBoard and their machine learning models.

4.2.2 Live-monitoring

Unlike the TensorBoard, the monitoring tools summarised under the term *Live-monitoring* were completely and from the ground up built by our team. For most of its visualisations the *matplotlib* [Hun07] library was used. The Live-monitoring tool combines two use cases:

First, it creates visualisations for all data recorded during the training, similar to those provided through the TensorBoard. This needs to be done to be able to quickly access the visualisations after the training has concluded, as the TensorBoard relies on abstract data files and needs to run on a local server in order to create visualisations. By taking the data we have at the end of the training and using our own visualisation tool, we create two types of diagrams: Scatterplots, which contain all samples for a certain parameter (see for example Figure 6.3 (c)) and lineplots, which show smoothed data, such as it would be available in the TensorBoard (see for example Figure 6.1).

Secondly, the tool simulates a market scenario identical to the one used during training an additional time. To understand why we do this, we need some additional information: during a training session, 'intermediate' models, as we will call them,

are being saved in regular intervals. These models contain the current policy of the agent and can be used the same as any other model of complete agents, the only difference being the quality of the agent, which can change over the course of a training run, both for the better and the worse. These intermediate models can then be used by a range of monitoring tools available to us. Since the models only contain the current policy of an agent, but not the history of states and actions preceding the model, we need to run separate simulations on these models to be able to analyse and evaluate them. For this we utilise our *Agent-monitoring* toolset (explained in detail in Section 4.3.1). In Section 6.2.1 we will discuss the results of a training session using the Live- and Agent-monitoring tools.

4.3 Monitoring complete agents

For monitoring trained RL and rule-based agents, we offer three major tools: The Agent-monitoring tool allows users to simulate a large number of episodes to visualise bigger trends, the Exampleprinter simulates a single episode, offering detailed insights into market states using an overview diagram, and the Policyanalyser is a static tool which can be used to analyse a vendor's reaction to different market states and competitor actions.

4.3.1 Agent-monitoring

The *Agent-monitoring* is a highly configurable tool for monitoring and evaluating different combinations of agents and marketplaces. Figure 4.1 shows how the tool works internally.

In addition to parameters provided to the marketplace and monitored agents, the following parameters can be used to configure the *Agent-monitoring* tool itself:

Episodes: This parameter decides how many independent simulations are run in sequence. At the start of each episode, the market state will be reset and randomised. Within an episode, vendors run through a configurable amount of time steps, during each of which they set prices (depending on the chosen economy type, this can range from only one price for new items to three prices, including a rebuy price for used items) and a set number of customers interact with them.

Plot interval: A number of diagram types enable the user to view averaged or aggregated data over a period of time. The plot interval parameter decides the size of these intervals. Smaller intervals mean more accurate, but also more convoluted data points. Computational time also increases with a smaller interval size.

Marketplace: Using this parameter, the user can set the marketplace on which the monitoring session will be run. Refer to Section 3.1 for an explanation of the different available marketplaces.

Separate markets: This parameter is a boolean flag that determines the way in which the monitoring session will handle the agents given by the user. If the flag is enabled, each agent will be initialised on a separate instance of the chosen marketplace, meaning that the agents will be monitored independent of each other. This functionality takes a lot longer than if the flag were disabled, as the whole marketplace is simulated once for each agent. While it may seem like the same results could be reached by simply starting multiple monitoring sessions with a

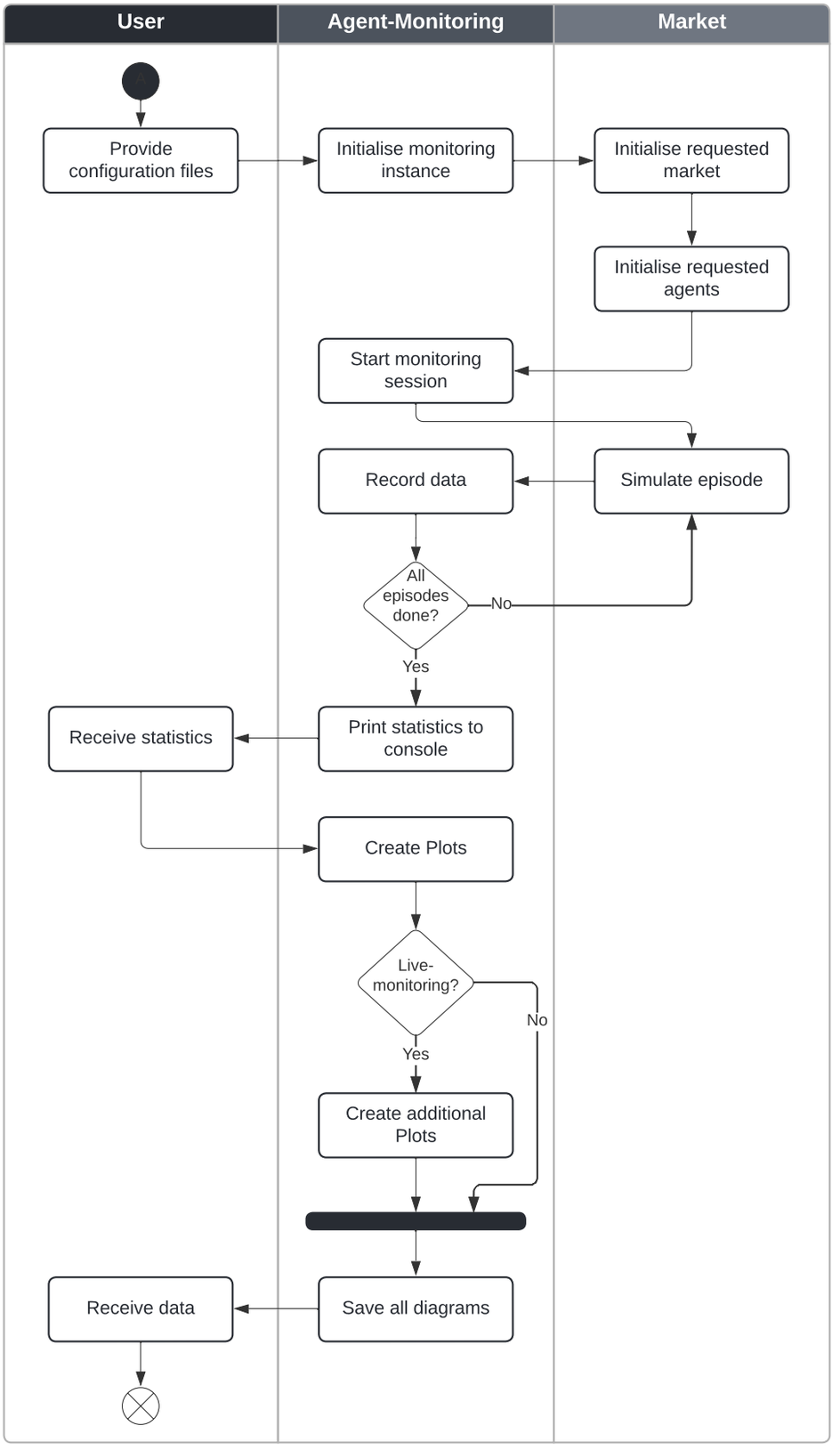


Figure 4.1: The internal workflow when running an Agent-monitoring session. Table A.3 lists the different types of diagrams created by both the Live- and Agent-monitoring and which of the recorded metrics they visualise.

different agent each, this is not the case. Using this flag instead, it is ensured that all agents get the exact same market states for each episode. By running multiple marketplaces in parallel using the *separate markets* flag, we can match the simulations as closely as possible. The most prominent use case where this flag is enabled is during the Live-monitoring after a training session, where all intermediate models are being monitored on separate markets.

If the flag is disabled, the monitoring tool will initialise only one marketplace and set the passed agents to directly compete against each other on this marketplace. This functionality is most useful when monitoring only a single agent, trying to determine its specific strengths and weaknesses against certain opponents, as it will complete a lot faster than if the flag were enabled.

Agents: Depending on the chosen marketplace, only a select number of agents can be chosen to be monitored, as each agent is built to interact with a specific type of marketplace. First off, all agents belong to one of the two major categories: *RL agent* or *rule-based agent* (for a more detailed overview see Section 3.3). RL agents can only be monitored on the marketplace type and market environment they were trained on, as these define the number of inputs and outputs the agent expects. Rule-based agents can only be used on the marketplace type they were built for, as each of them makes assumptions about the number of prices they will need to set, but the market environment may be freely chosen. This leads to not all marketplace types having the same amount of rule-based vendors available. Following their importance for our simulation framework, the Linear Economy has the least and most often weakest vendors available, while the more refined competitors are most of the times only available as a version compatible with the Circular Economy with rebuy prices.

During each episode and for each vendor, all market events are being recorded. At the end of the monitoring session, the collected data is evaluated in different visual formats. First of all, all data that would be available to see in the *TensorBoard* during a training session is visualised using density plots. These plots can be used to compare the vendors in a more granular way, if for example the effect of a parameter on the customer's sell-back behaviour of used items should be tested. Another visualisation that is created is a histogram containing the cumulative profits per episode for each agent, plotted against each other (see for example Figure A.11 (b)). This allows for a quick overview to see which agent had an overall better performance. One additional type of diagram is only created if the Agent-monitoring is run through the Live-monitoring tool: Violinplots. These plots, which are created for all data points available through TensorBoard (also see Table A.3), depict distributions using density curves, accentuating the minimum, maximum and median values. Violinplots are used by us to compare different training stages of an agent, as small policy changes can have great impact on these values. For exemplary Violinplots created after training, see Figure 6.4.

Aside from monitoring after a training session, a common use case of the *Agent-monitoring* tool is to test trained agents against competitors other than the ones it was trained against. This is done to test the agent's capacity to adapt to different circumstances, an important factor in deciding an agent's quality, as its competitors in the real market will differ from any it has encountered in training, due to the sheer vastness of options when it comes to dynamic pricing models available nowadays.

4.3.2 Exampleprinter

The *Exampleprinter* is a tool meant for quickly evaluating a market scenario in-depth. When run, each action taken by the monitored agents is being recorded, in addition to market states and events such as the number of customers arriving and the amount of products thrown away. At the end of this quick simulation, an animated overview diagram is created, which shows all actions and their consequences for each step in the simulation, see for example Figure 6.5. Due to the large amount of data that is being collected and visualised and the overhead would come with doing so for hundreds of episodes, we chose to disconnect this functionality from large-scale tools such as the Agent-monitoring. While the Agent-monitoring tool could be seen as a tool that imitates Macro-economic behaviour, simulating hundreds of days through hundreds of episodes, the Exampleprinter instead simulates only one day, recording and visualising all data collected during that time.

4.3.3 Policyanalyser

The last tool we want to introduce is the *Policyanalyser*. The *Policyanalyser* is our only tool which does not simulate a market. Instead, the tool can be used to monitor an agent's reaction to different market states. The user can decide on up to two different features to give as an input, such as a competitor's new and refurbished prices, and the Policyanalyser will feed all possible input combinations to the agent and record its reactions. When initialising the *Policyanalyser*, the user defines a number of parameters: The agent whose policy should be analysed, as well as the marketplace and the competitors that should be used, just as is done for all the other tools. Additionally, the user defines a **template market state**, a market state containing all values that are passed to the analysed agent, such as the number of items currently in circulation and the prices of competitors. Lastly, a list of **analysed features** needs to be provided, which defines one or two features of the template market state that should be varied. When the *Policyanalyser* is run, these features are inserted into the template market state, overwriting the initial values and creating a new combination. This new market state is then passed on to the *policy*-method of the analysed agent (example policies of some of our rule-based agents can be found in Section A.3), and its reactions are recorded and visualised. See Section 6.2.4 for use cases of this tool.

The *Policyanalyser* is the monitoring tool which operates on the smallest scale out of all the tools we built for our framework. It allows users to define any market state they want and to then accurately monitor a vendor's reactions to changes to this specific state. While the tool can just as well be utilised to test new rule-based strategies, it is very much meant to be used as a way to understand RL agents better, as their policies are not immediately visible to the end-user and must therefore be discovered through tools such as the one's we built.

5 The *recommerce* Workflow

The main goal of the market simulation framework is to provide a simple-to-use but powerful tool for training RL algorithms on highly configurable markets for users in both a research and a business context. To achieve this, multiple components had to be developed and connected to create the workflow we now provide. This chapter will introduce the most important parts of the workflow.

When working with our simulation framework, users can choose from two options: First, it is possible to use our tool via a custom Command line interface (CLI). Alternatively, users can interact with the framework through a web-interface, which utilises Docker for remote-deployment of tasks issued by the user. For detailed insights into our web-interface and remote deployment, see [Her22].

Figure 5.1 depicts the common workflow activities in our framework. For all possible tasks, the user needs to provide configuration files, which define both the task to be worked on and parameters which influence market and agent behaviour (see Section 5.1). After the configuration files have been validated, the simulation framework initialised the requested marketplace and agents, and then starts the requested task, for which there are currently three options provided through the CLI or web-interface: *Training*, *Agent-monitoring* and *Exampleprinter*. You may have noticed that one of our monitoring tools is missing from this list, the *Policyanalyser*. As is mentioned in Section 7.2.4, this tool must unfortunately still be started manually by the user, as it has not been integrated into the rest of the workflow yet.

At the end of the respective task, the user is always provided with the various diagrams and data (e.g. trained RL models) collected during the respective task, which can then be used in subsequent tasks.

5.1 Configuring the run

Configuration is one of the most important aspects of the workflow. Without it, each simulation and training session would produce similar, if not the same results. By tweaking different parameters of a run, market dynamics can be changed and agent behaviour and thereby performance be influenced. The goal of our monitoring tools is to enable users to assess the extent to which each parameter influences certain characteristics of the training and/or monitoring session, and to enable them to make informed decisions for subsequent experiments.

Ultimately, all configuration is done using various `.json` files which contain key-value pairs of the different configurable items. We further differentiate between different groups of configurations, which means that hyperparameters influencing the market, such as maximum possible prices or storage costs, are being handled separate from parameters needed for RL agents, such as their learning rates, allowing users to easily change and tweak parameters involving different parts of the framework. Examples of such configuration files can be found in Section A.4.1.

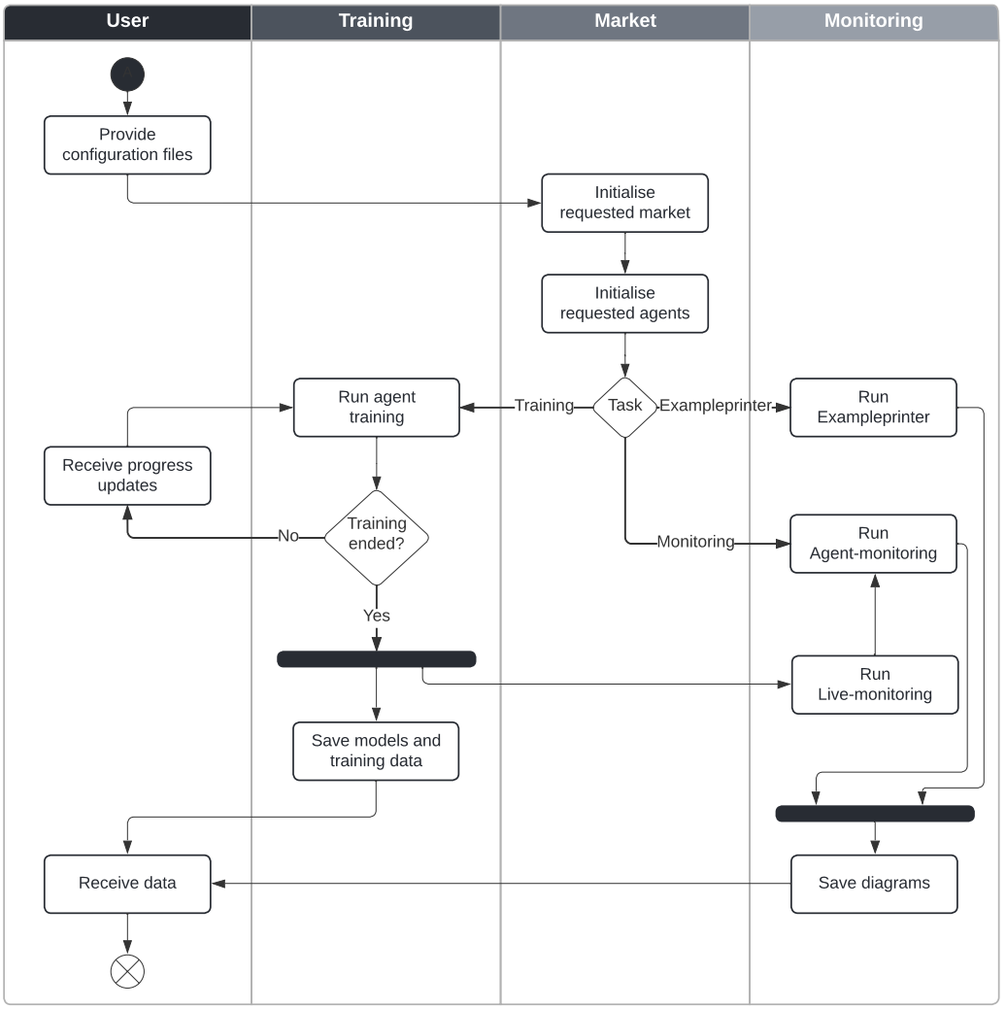


Figure 5.1: Diagram depicting possible workflows without webserver interaction.

5.2 The monitoring workflow

Within the recommerce workflow, there are two points in time when our monitoring tools are or can be used. While a training session is running, the TensorBoard tool is automatically used to record metrics and give insights into the current training run, by creating visualisations of current market states. By saving intermediate models, the Live-monitoring enables the user to compare agent models saved at different times within the training process, after it has concluded. This can be especially useful when trying to determine the most effective amount of training steps after which a model does not improve further, to optimize future runs. After the training has finished, or at any point disconnected from a training session, our tools described in Section 4.3 can be used to further analyse the trained models. If a training session is terminated by the user before it has completed, the Live-monitoring tool will not be run, but all intermediate models saved up to that point can still be used to monitor the agent's policy at that point in time. The next chapter, Chapter 6, is dedicated to the monitoring workflow, where we will first run a training session, and then monitor and evaluate the results we get from it.

6 Monitoring an Experiment

In this chapter, we will put the tools and workflows described in the previous sections to the use. We will train an RL agent, and then monitor it using all the tools at our disposal, highlighting where each tool is most useful, and what could be improved.

6.1 Setting up the experiment

Before starting our monitoring, we will need to conduct an experiment, where an RL algorithm is being trained on a market environment. For our experiment, we will train an RL agent using the SAC-algorithm [Haa+18] on a Duopoly marketplace with rebuy prices. The agent will be trained playing against a rule-based agent, more specifically the *RuleBasedCERebuyAgentCompetitive*, as presented in Section 3.3.2. The configuration files for this experiment can be found in Figure A.4, Figure A.5 and Figure A.6. We will refer to this experiment as the *SAC-Duopoly* experiment. To ensure that we are evaluating an agent with a performance that is to be expected with the provided parameters, we will conduct the experiment multiple times to be able to identify outliers in the data. All diagrams except those shown in Figure 6.1 (which contains diagrams from each of the four experiment runs) have been taken from the same experiment run, denoted as *SAC-Duopoly_1* in Figure 6.1 (a).

For the interested reader, a number of diagrams created through a second experiment are available in the Appendix under Section A.5.2. In this second experiment, a PPO-Agent (see Section **??**) was trained against four rule-based competitors on a Circular Economy market with rebuy prices. The configuration files for this experiment can be found in Section A.5.1.

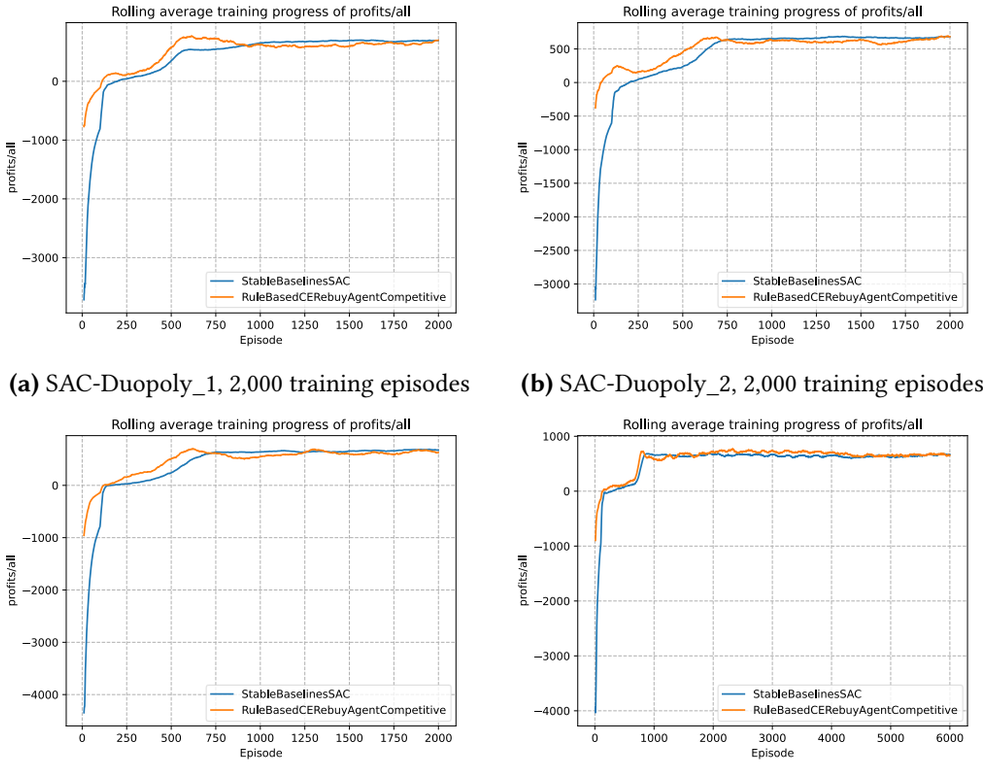
6.2 Experiment results

In the following sections we will use our different monitoring tools on the results of the SAC-Duopoly experiment. We will start with the Live-monitoring tool, which runs automatically after the training run has completed and creates over 90 graphs and diagrams already. Due to this high number of available diagrams, we will always only look at a curated selection, highlighting those which give the best insights into the trained agent.

6.2.1 Live- and Agent-monitoring

This section will focus on the diagrams created by the Live-monitoring tool after training, which always runs an Agent-monitoring session as well, to immediately provide the user with many additional useful diagrams without the need to run the tool manually.

A commonly asked question when deciding on the quality of an RL agent is their *stability*. If an algorithm is stable, the trained agent will produce similar results



(a) SAC-Duopoly_1, 2,000 training episodes

(b) SAC-Duopoly_2, 2,000 training episodes

(c) SAC-Duopoly_3, 2,000 training episodes

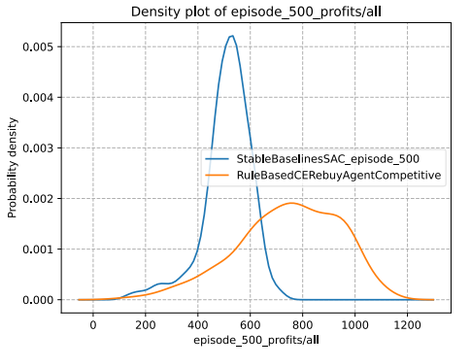
(d) SAC-Duopoly_4, 6,000 training episodes

Figure 6.1: Profit per episode of four different training runs of an SAC-Agent on a Duopoly market.

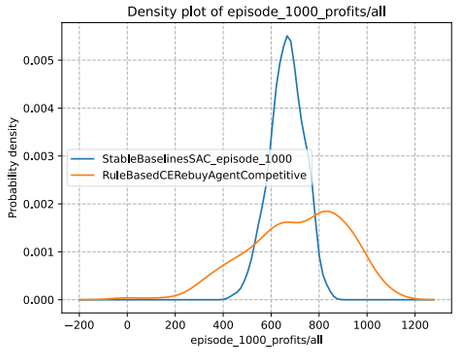
over multiple training sessions, on the condition that the parameters do not differ. Not only the rewards achieved at the end of the training will be very similar, but also the amount of episodes needed to reach certain thresholds. In the case of the SAC-Duopoly experiment, we ran the same configuration four times: The first three experiments were run using the exact same parameters, for the fourth experiment the amount of training episodes were tripled, meaning that the SAC-Agent had more time to alter its policy. Figure 6.1 shows the results of these four training sessions, created using the Live-monitoring tool and visualises the data collected during the training process. Although many other graphs are created (Table A.3), the visualisation of the total profit of the agent is the most convenient to use when evaluating an agent's performance, as the total profit is the parameter which the agent is trying to optimize.

Figure 6.1 shows the stability of the SAC-Agent very well. The agent not only reached the break-even threshold of a reward of 0 after around 150 episodes in each of the four training runs, but no matter the total length of training (see the model in Figure 6.2 (d), which was trained three times as long as the others), the profit would always stabilize and stay at around 670. Had the monitoring tools shown that the agent performs worse than expected in some of the experiments, we could conclude that this particular algorithm is not fit for the type of work required by our market simulation.

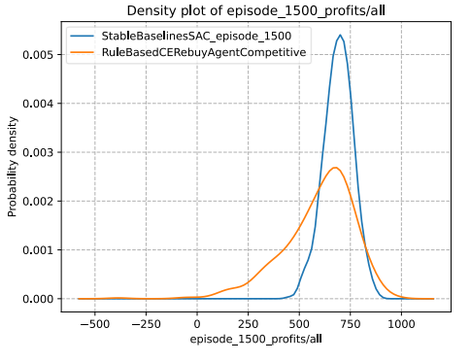
We can also observe that the profits of the SAC-Agent and the rule-based agent (in the case of this experiment, a *RuleBasedCERebuyAgentStorageMinimizer*) seem to be closely linked to each other in this particular scenario. In the beginning of each experiment, when the RL agent still knows very little about the market and makes



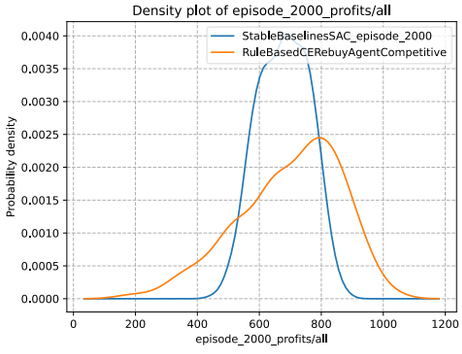
(a) Model trained for 500 episodes



(b) Model trained for 1,000 episodes



(c) Model trained for 1,500 episodes



(d) Model trained for 2,000 episodes

Figure 6.2: Probability densities for achieving a certain profit for four different training stages of the model trained during the *SAC-Duopoly_1* experiment.

great losses, the rule-based agent also has a hard time to perform well. However, as soon as the SAC-Agent starts to perform better, the rule-based agent is also able to increase its mean profits at around the same rate as the SAC-Agent. Even more interestingly, the agents not only increase their profits at the same rate, but also very quickly arrive at a point where they earn the same mean amount as the other.

This might lead one to believe that the two competitors' policies align closely with each other. To validate this claim, another type of diagram created by the Live-monitoring tool can be used: The densityplots. These diagrams visualise probability densities for the various datasets recorded. While the diagrams shown in Figure 6.1 simply visualise data that was collected during the training run, densityplots are created by running the Agent-monitoring tool, where the marketplace is simulated an additional time. This allows us to use the 'intermediate' models we saved during the training run (see Section 4.2.2) to compare the RL agent's policies at different points in time during the training.

Figure 6.2 shows the densityplots of profit-per-episode for four different training stages of the *SAC-Duopoly_1* experiment. From this it can be concluded that the claim that the two competitors' policies align closely is incorrect: even though the mean reward within an episode is always very similar (Figure 6.1 (a)), the SAC-Agent achieves rewards which lie closer together, while the rule-based agent's rewards have more of a spread.

We can also observe that the models which were trained for longer are not necessarily better or even the same quality of the models with less experience. There is a significant improvement going from the model shown in Figure 6.2 (a) to the one in Figure 6.2 (b), this shift in the probability density curve can be correlated

with the maximum mean rewards the two models could achieve during the training: For the model trained for 500 episodes, this was around 450, for the other it was about 660 (Figure 6.1 (a)). Both of these values have respectively high probabilities of being reached during the second simulation after the training has concluded, i case of the model trained for 1,000 episodes, this even coincides with the maximum in the density plot (Figure 6.2 (b)). Going from the model trained for 1,000 episodes to the next one, which was trained for 1,500 episodes (Figure 6.2 (c)), both the probability densities and the mean rewards stay very close to each other. From this we can conclude that training the SAC-Agent for more than 1,000 episodes is very likely to not have a great effect on the maximum reward achievable by the algorithm. Going from the model trained for 1,500 episodes to the last one, saved at the end of the training (Figure 6.2 (d)), we can however see a deterioration in performance: While the mean rewards hardly changed (see Figure 6.1 (a)), the probability density curve got wider at its base, extending out further towards a reward of 400, and lowering the probability of achieving a reward of 700 from previously above 0.5% to under 0.4%. This means that the model which was trained for the longest time produces less predictable results than those trained less, which is a tame version of so-called *Catastrophic Forgetting*, which will be explained further in Section 7.2.1. The insights gained by our monitoring tools combined with the fact that we save 'intermediate' models of the RL agent during training allows us to find the optimal trained model to use for further investigation and eventual deployment in the real market. In the case of the *SAC-Duopoly_1* experiment, the optimal model would be the one trained for 1,000 episodes.

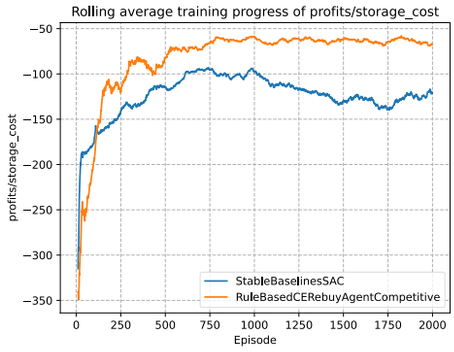
6.2.2 Further investigation

Besides the mean profits achieved during training, our Live-monitoring tool offers many other useful diagrams as well, a selection of which will be shown in this section. The graphs used will be from the *SAC-Duopoly_1* experiment.

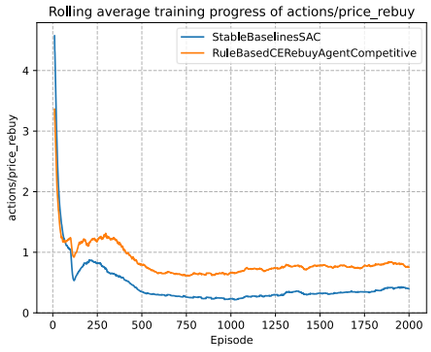
Users may ask themselves how the profits achieved by the different vendors are split between the two available retail channels of new and refurbished products, how many products were bought back from customers or how much the vendors had to pay for storage of these used products. For all of these questions, the Live-monitoring (together with the Agent-monitoring) tool offers two types of diagrams: First, simple lineplots as shown in Figure 6.1 can be used, as well as scatterplots which visualise the exact data recorded during the episode, instead of the trends shown by the lineplots. Figure 6.3 shows a number of different metrics, all of which are available as both linepots and scatterplots (see Table A.3 for a list of all metrics and visualisations).

Many connections can be made when evaluating different diagrams side-by-side, such as the observation of the initially high storage costs of both vendors in Figure 6.3 (a) being caused by high rebuy prices set by the agents (Figure 6.3 (b)). High rebuy prices make it likely that customers are willing to sell back their products, which leads to (over)full inventories and high storage costs. High storage and rebuy costs will have then caused a policy change to set lower rebuy prices, thereby de-incentivising customers to sell back as many products as they used to and lowering the agent's storage costs, increasing the profit margin.

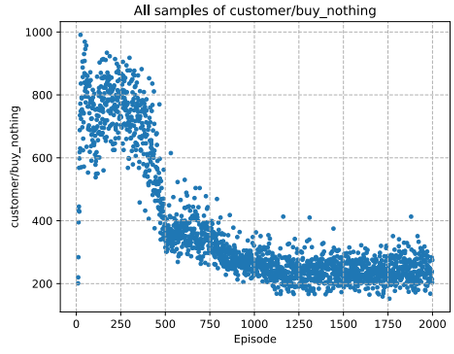
The next pair of diagrams shows the connection between a high number of customer that buy nothing, as visualised in Figure 6.3 (c) correlating to high prices for



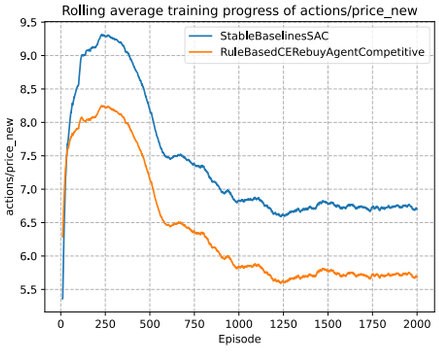
(a) Storage costs for bought back products



(b) Rebuy prices set by the vendors



(c) Customers that bought nothing



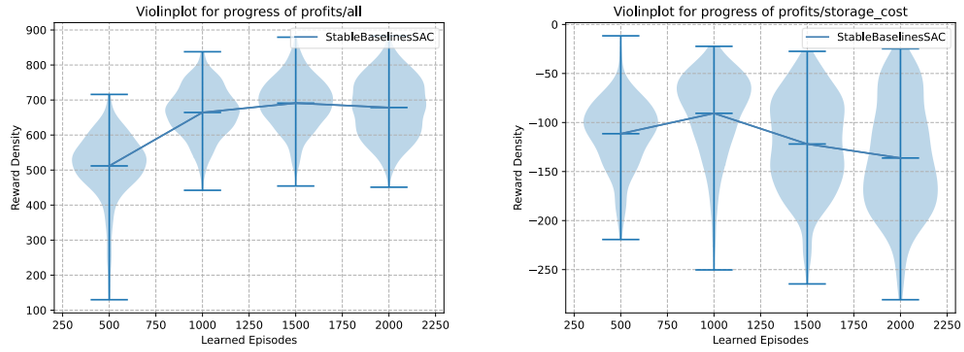
(d) New prices set by the vendors

Figure 6.3: Diagrams visualising various data points collected during training of the *SAC-Duopoly_1* experiment.

new products, shown in Figure 6.3 (d). As mentioned in Section 3.2, the customers implemented in our framework make their purchase decisions based on the prices they are presented with, which leads to many customers choosing to buy nothing before paying prices as high as set by the vendors.

If the Agent-monitoring is run after a training session (with a number of intermediate models), it also creates violinplots for all of the collected metrics (Table A.3). A selection of such diagrams can be found in Figure 6.4, showing the respective total profits (Figure 6.4 (a)) and storage costs (Figure 6.4 (b)) for each intermediate model of the trained SAC-Agent. As explained in Section 4.3.1, these plots visualise the probability distributions as shown in Figure 6.2 in a more condensed way, providing additional data such as actual maximum, minimum and median values as well. The biggest upside of the Violinplots is however that they are able to show the distributions for all intermediate models in one diagram, which allows for even better comparisons. Looking at Figure 6.4 (a) we can immediately see the difference in the spread of total profits achieved between the models trained for 1,500 and 2,000 episodes respectively, for which we previously needed to consult and compare two different diagrams (Figure 6.2 (c) and Figure 6.2 (d)). Similarly, Figure 6.4 (b) is able to tell us something else we did not know before: the longer a model was trained for, the more likely it is that it will induce high storage costs, a trend which was not necessarily visible in Figure 6.3 (a).

Violinplots do however not replace the need for densityplots, as both have an equally useful way of displaying data. While the violinplots show rough distributions plotting against the real numbers on the y-axis, the densityplots show the concrete probability values for each possible data point.



(a) Total profits for all intermediate models

(b) Storage costs for all intermediate models

Figure 6.4: Violinplots showing a selection of collected data when running Agent-monitoring after training.

6.2.3 Exampleprinter

All of the diagrams shown in the previous section were created as part of the automatic monitoring done after a training session has concluded. There are however two other tools at our disposal to monitor the trained RL agent after they have been trained. For this, we only need the intermediate models saved during and at the end of training. In the case of the *SAC-Duopoly_1* experiment, we will be using the model saved after 1,000 episodes, as we discovered it had the best performance, see Section 6.2.1. The first tool we will now utilise is the Exampleprinter, as introduced in Section 4.3.2.

To configure this monitoring run, we will again need three configuration files: one defining the task to be done (for which we will re-use the one shown in Figure A.4, only exchanging the 'task' keyword), and one for both the hyperparameter-configuration of the market and the SAC-Agent, for which we will also re-use the configuration files used for training (Figure A.5 and Figure A.6). By using the same configuration files again, we can emulate the market the agent was initially trained on as closely as possible.

At the end of the monitoring session, we receive an animated overview diagram that cycles through all time steps, allowing us to identify and examine potentially interesting time steps in the simulation. Due to the nature of this diagram being animated, we will not be able to thoroughly examine the results of this monitoring tool in this thesis. Instead Figure 6.5, which shows the 17^{th} time step of the simulation, can be used to gain an understanding of the information that can be gained from this tool.

6.2.4 Policyanalyser

After an RL agent has been succesfully trained, or after implementing a new rule-based pricing method, users may want to analyse specific characteristics of the agent's policy using the Policyanalyser. We will show both use cases of the Policyanalyser, by first analysing the policy of our trained SAC-Agent and then analysing the policy of the rule-based *RuleBasedCERebuyAgentStorageMinimizer*. As was explained in Section 4.3.3, we need to define both a marketplace and a template market state before running the Policyanalyser. For our experiments, we used a Circular Economy with rebuy prices, and the following default market state:

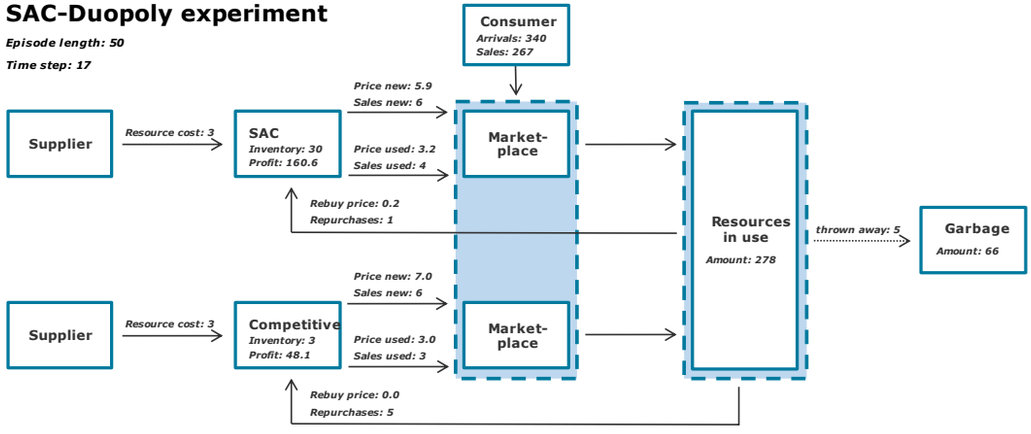
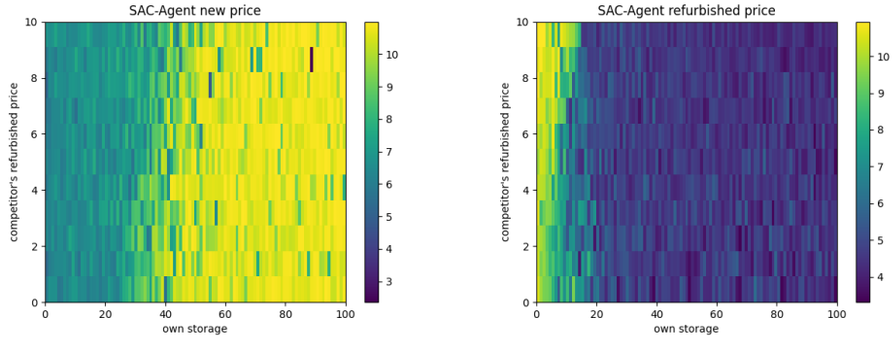


Figure 6.5: Actions and market states during step 17 of the Exampleprinter session.

- Number of products in circulation: 75
- Competitor's price for new products: 5
- Competitor's price for refurbished products: 3
- Competitor's rebuy price: 2
- Number of items in own storage: 10
- Number of items in competitor's storage: 12

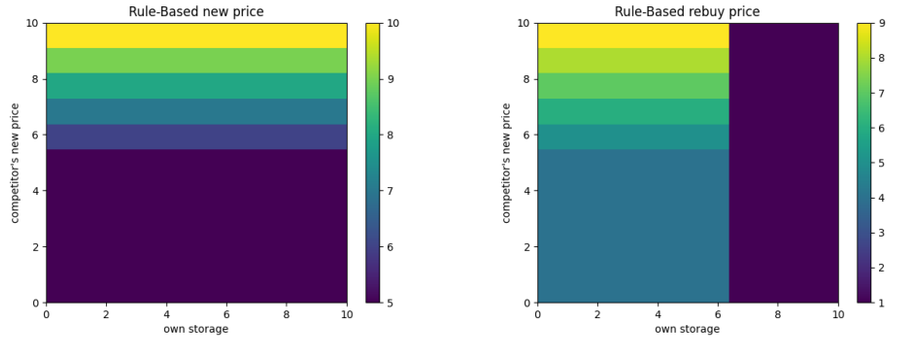
In each run, the Policyanalyser will then replace two of those values, as specified by the user, with the features that should be analysed. For our first use case, we will analyse the policy of our trained SAC-Agent. The features that should be replaced are the SAC-Agent's own storage, which will range from 0 to 100, and the competitor's price for refurbished items, ranging from 0 to 10. The results of combinations of features for the agent trained during the SAC-Duopoly experiment can be found in the Appendix under Section A.4.2, as we can only analyse a limited number of diagrams in the main part. Note that it does not matter which policy the competitor follows, as the tool does not simulate the market, but only pass market states to the monitored agent. Figure 6.6 shows the respective new and refurbished prices the SAC-Agent would set upon receiving these specific market states.

From Figure 6.6 (a) we can infer that the SAC-Agent is not very likely to ever set new prices lower than 6, which coincides with the observed prices set during



(a) New prices **(b)** Refurbished prices

Figure 6.6: Prices set by the trained SAC-Agent, depending on the competitor's refurbished price and the agent's own storage.



(a) New prices **(b)** Rebuy prices

Figure 6.7: Prices set by the rule-based agent, depending on the competitor's new price and the agent's own storage.

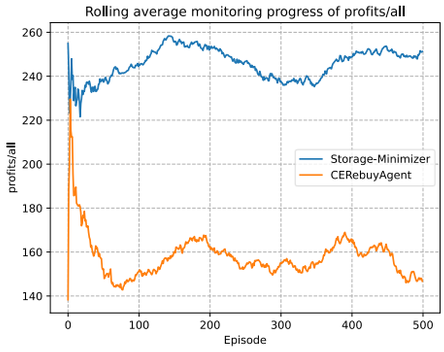
training, see Figure 6.3 (d). Figure 6.6 (a) also shows that the SAC-Agent will increase prices for new items if it has a higher number of items in its storage, but not necessarily if the competitor's price for refurbished items rises. By increasing the price for new items, the agent tries to de-incentivise customers to buy those items from it. At the same time, when looking at Figure 6.6 (b), the SAC-Agent will also decrease the price for refurbished items if it has more in storage, again to incentivise customers to buy those products. From both of these diagrams we can see that the SAC-Agent prioritizes making pricing decisions based on the amount of items in its storage, and less based on competitor prices.

Next, we will analyse the policy of the *RuleBasedCERebuyAgentStorageMinimizer* using the Policyanalyser. Figure 6.7 shows the results of this monitoring session. Figure 6.7 (a) shows the prices set by the vendor for new products, which follows the agent's policy implementation (see Figure A.2) of always undercutting competitor prices by 1, but always sell above production price (which in this case is 3, see the market configuration file in Figure A.5).

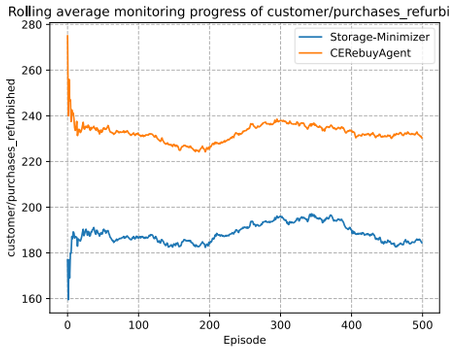
Figure 6.7 (b) shows the price the rule-based agent will set for buying back items from customers. According to its policy, the rebuy price depends on both the agent's own storage and either its own price for new items (if inventory is low) or competitor's rebuy prices, both of which can be seen nicely in the diagram. As soon as inventory exceeds the limit set in the policy, the agent sets a rebuy price that is 1 lower than the lowest competitor's rebuy price - which in this case is 2, as defined by the template market state. If inventory is low enough, rebuy prices are instead dependent on the vendor's own new prices, as seen in Figure 6.7 (a), always being one lower, also following the competitor's new price.

6.2.5 Other use cases for Agent-monitoring

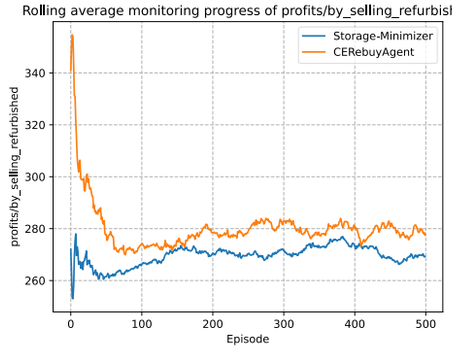
Even though most of the diagrams that are created through the Agent-monitoring tool are created if it is run immediately after a training session, it is disconnected from the Live-monitoring tool. There are two major reasons for this, explained in the following two sections.



(a) Total profits per episode



(b) Refurbished products sold per episode



(c) Profits by refurbished sales per episode

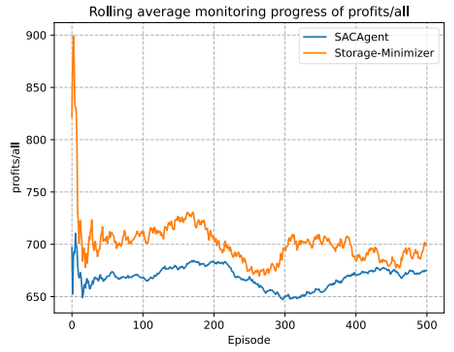
Figure 6.8: Diagrams created during an Agent-monitoring session comparing two rule-based pricing methods.

Testing rule-based pricing methods

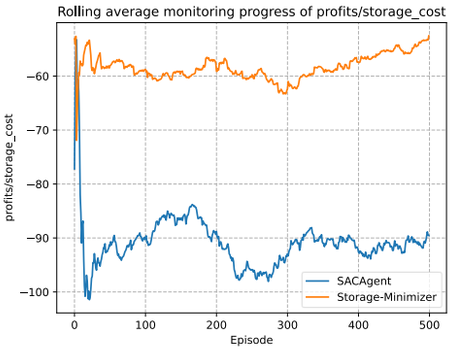
First, aside from training RL agents, even though this is the main focus, our market simulation framework can just as well be used to implement and test classically rule-based pricing methods, as we have done ourselves for our rule-based vendors. In order to make this workflow possible, users need to be able to use the Agent-monitoring tool, as it is the only way of running large-scale simulations of different marketplaces aside from training an agent, which is not an option for this use case. Figure 6.8 shows the results of an Agent-monitoring session with a *RuleBasedCERebuyAgent* playing against a *RuleBasedCERebuyAgentStorageMinimizer*, a *Data-driven model* that tries to keep the amount of products in its storage as low as possible. Its policy can be found in Figure A.2. As was already mentioned in Section 3.3, we can see that the *RuleBasedCERebuyAgent* performed significantly worse than the other rule-based agent (Figure 6.8 (a)). This is mainly due to the fact that it is a purely *Inventory-based model*, while the *RuleBasedCERebuyAgentStorageMinimizer* is a *Data-driven* model. Certain characteristics of the *RuleBasedCERebuyAgent-StorageMinimizer* can also be seen in Figure 6.8 (b) and Figure 6.8 (c), as it sells significantly less refurbished products, but is still able to make about the same amount of profits as the simpler *RuleBasedCERebuyAgent*. This can be attributed to its quality of keeping storage costs low by minimizing inventory size.

Testing trained models against different competitors

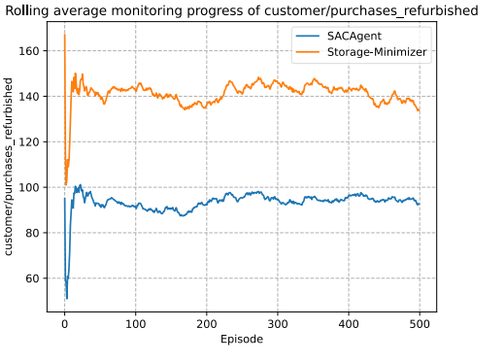
The second reason for having the Agent-monitoring as a stand-alone tool is to be able to monitor trained RL agents not only with the competitors they were trained



(a) Total profits per episode



(b) Storage costs per episode



(c) Sales of refurbished items per episode

Figure 6.9: Diagrams created during an Agent-monitoring with the SAC-model saved after 1,000 training episodes playing against a *RuleBasedCERebuyAgentStorageMinimizer*.

against, but any combination of other agents, including other trained RL agents. Figure 6.9 shows this use case on the example of our trained SAC-Agent from the SAC-Duopoly experiment playing not against the *RuleBasedCERebuyAgentCompetitive* it was trained against, but instead playing against a *RuleBasedCERebuyAgentStorageMinimizer*, as introduced in the previous section.

From Figure 6.9 (a) we can infer that the SAC-Agent we trained does not play as well against the *RuleBasedCERebuyAgentStorageMinimizer* as it did against the *RuleBasedCERebuyAgentCompetitive* (Figure 6.1 (a)), as it makes slightly less profits overall and is consistently being outperformed by the rule-based agent. This can be expected, as the trained agent no longer learns the behaviour of its competitor, but now acts according to the previously learned policy. This means that with a change of the competitor, the learned policy may not be as good as it was against the previous opponent.

The impact the different rule-based agent has on the simulation can also be seen in the difference between storage costs of both vendors between the two monitoring runs. While in Figure 6.3 (a), both vendors paid storage costs between 50 and 150, this was not the case for the second simulation, where storage costs were much lower for both vendors (Figure 6.9 (b)). As the *RuleBasedCERebuyAgentStorageMinimizer* focusses on keeping a low inventory at all times, it also buys back less items. This seems to have agreed with the SAC-Agent's policy, as the trained agent also kept fewer items in inventory in the simulation against this competitor. It was however not able to manage inventory as well as the rule-based agent, as both its storage costs were higher, and sales of refurbished items lower, see Figure 6.9 (c).

7 Outlook & Summary

In this final chapter, we will look back at our framework and give an outlook on how different parts of it, especially the monitoring tools, could be improved in the future. This will be followed by a short summary of the learnings from this thesis.

7.1 Modelling a realistic *recommerce* marketplace

We do not claim in any way that our simulation framework is exhaustive or complete. This section will focus on ways in which different parts of it could or should be improved in the future, to model a more realistic marketplace.

In the modern recommerce market, more and more consumers make their initial purchasing decisions with the product's eventual resale value already in mind [TP19]. Additionally, a great number of different motivations for choosing second-hand or refurbished products over traditional new ones can be identified. An exemplary study conducted in 2008 [RG08] classified such motivations into 15 different categories (see Table A.2), all of which could be used to add dimensions to customer behaviour in our simulation framework, thereby making the whole simulation more realistic.

Our current limited customer behaviour results in the fact that our framework can not be used out-of-the-box for any kind of market. However, great care was taken during the development process to build each part of the simulation in a modular way, so that new functionality can easily be integrated into it. Marketplaces, customers and vendors (rule-based as well as those using Reinforcement learning) have all been implemented as classes disconnected from each other, so that new variants of each can easily be added to the pool of available options to be used in experiments. The same goes for our monitoring tools, all of which have been built to work with any (valid) combination of input parameters. For reference, all of the classes shown in Figure 3.1 can be extended or replaced with ease. Please also refer to [Dre22] for more information on the modularity of our framework.

The lack of realism on certain areas of the framework does not directly impact the way that our monitoring tools work or can be used. It must however be noted that the interpretation of results must always be based on the knowledge of these limitations.

7.2 Improving our monitoring tools

While the previous section focussed on way in which our simulation framework could be extended to be more realistic, this section will instead focus on ways in which our various monitoring tools could be improved. While all of the tools we currently have available are able to do what is expected of them, and they each have their own strengths, there are still a number of ways that the different tools could be enhanced.

7.2.1 Live-monitoring

Currently, the Live-monitoring tool works by first taking and visualising all the data collected during a training run, and then running the Agent-monitoring tool on the saved intermediate models. The biggest downside to this is that users need to wait until the training session has finished before the Agent-monitoring is run, meaning that a lot of potential information is lost. Imagine a scenario where a training session was initialised to run for 10,000 episodes, saving intermediate models each 1,000 episodes. In the end, when the Live-monitoring tool is run, the user may find that the best model was the one saved after 3,000 episodes, meaning that a lot of time was wasted training an additional 7,000 episodes. The possibility of this happening may at first seem counterintuitive, but is a common phenomenon when training RL agents, known as *Catastrophic Forgetting* (see also [Cah11]). By improving the Live-monitoring tool with an option to **run the Agent-monitoring whenever an intermediate model is saved**, the user would be able to recognize such trends much faster and terminate the experiment at the right time. This feature could be further enhanced with a smart built-in option that **terminates the experiment for the user if a downward trend in performance is detected**. Specific thresholds for these terminations should also optionally be **set by the user**. For this, the data created during the Agent-monitoring tool would need to be saved in a machine-readable form - graphs and diagrams are not useful here.

7.2.2 Agent-monitoring

This brings us to improvements that could be made to the Agent-monitoring tool. At the moment, a lot of data is recorded when simulating the marketplace, but only graphs and diagrams are created as a result of the simulation. By **giving users the option to have data saved as (for example) `.csv` files**, users would be enabled to use the results of the simulation in other ways more easily, even for monitoring and evaluation tools completely disconnected from our own framework and the tools we provide. Reproducibility is also a major concern when it comes to evaluating simulation results, as was already mentioned in Chapter 2 and is discussed in many papers such as [Hen+17] and [Isl+17]. In our simulation framework, with the start of a new episode the market state is always shuffled randomly, to allow RL algorithms to properly explore the environment. This is however creating the problem of creating simulations which are currently impossible to reproduce, which could be solved by **introducing a seed-based system for shuffling market states and sharing this seed with the user**. When running a different simulation with the same seed, assuming that the marketplace type and environment stay the same, users can recreate the same random market states that are set at the start of an episode, allowing for even better and in-depth comparisons of different vendors, past a single run of the Agent-monitoring tool.

7.2.3 Exampleprinter

The Exampleprinter is a great tool for quickly monitoring and evaluating a certain market setup. However, only for one specific combination of marketplace type and market environment, a Duopoly scenario of a Circular Economy with rebuy prices, the animated overview diagram is created. **Building templates and adding dia-**

gram support for more scenarios should therefore be a priority when enhancing the Exampleprinter. Additionally, the **market-seed feature** introduced in Section 7.2.2 should also be implemented for the Exampleprinter, to allow users to run a simulation multiple times to monitor possible discrepancies in agent behaviour and find outliers in the data. This could be further enhanced by adding a configuration option that allows for **more than one episode to be simulated at a time**.

As a combined enhancement for both the Agent-monitoring and the Exampleprinter, the **Exampleprinter could be integrated into the Agent-monitoring**, while still being its own tool, the same as the Agent-monitoring is integrated into the Live-monitoring.

7.2.4 Policyanalyser

The biggest and most important improvement to the Policyanalyser does not concern its concrete features, but the way users interact with it. Currently, all of the other monitoring tools are either integrated into some part of the workflow (e.g. the Live-monitoring), or can easily be started using simple commands, see Chapter 5. This is however not the case for the Policyanalyser, which must be started by going into the code itself, which is less than ideal from a user-perspective. So, **integrating the Policyanalyser into the workflow**, both by **creating a user-facing interface** for it and by **adding configuration options to start it after a training session** are features that should be a priority when continuing work on the framework. Users are also able to use the Policyanalyser to analyse a large number and combination of features, so **curating a list of useful feature combinations to be analysed** would aid many users when using this tool.

7.3 Summary

Using the market simulation framework that was built within the scope of the bachelor's project, users can simulate complex recommerce market situations. Thanks to the modular nature of the framework, different components, such as customer behaviour, can be updated, exchanged or added upon to create a configuration that fits the individual use case. The goal of such simulations is to enable users to implement, test and evaluate various dynamic pricing methods. While classically rule-based pricing methods play a big part in the framework, the majority of implemented and tested pricing methods are those based on RL algorithms, a machine learning technology. The goal of using such algorithms is to automate and optimize the dynamic pricing problem in the recommerce market. By using our framework, users are provided with a large number of tools to monitor, compare and evaluate any combination of pricing methods, enabling them to find the right fit for their needs. The provided tools work on many different levels, from those simulating large amounts of episodes, allowing for an analysis of potential macro-economic implications following specific approaches, to those that work on th smallest possible scale, testing an agent's pricing policy against every possible combination of market states and competitor actions. This allows for a thorough investigation of different strengths and weaknesses of the monitored pricing agents, a necessary prerequisite before being able to employ them in the real market and giving them power over actual pricing decisions.

Bibliography

[Aba+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu and Xiaoqiang Zheng. **Tensorflow: large-scale machine learning on heterogeneous distributed systems**. *Corr* abs/1603.04467 (2016). arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467 (see page 16).

[Arc05] New York Times (Archived). *As i.t. goes, so goes forrester?* Accessed: 2022-06-21. 2005. URL: https://web.archive.org/web/20180613040854/https://www.nytimes.com/2005/02/18/business/yourmoney/as-it-goes-so-goes-forrester.html (see page 1).

[Bas22] Stable Baselines3. *Rl algorithms*. Accessed: 2022-06-06. 2022. URL: https://stable-baselines3.readthedocs.io/en/master/guide/algos.html (see page 14).

[Bes22] Nick Bessin. **The marketplace of the future: simulation of market processes in re-commerce**. Hasso-Plattner-Institute, 2022 (see pages 7, 8).

[Bro+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang and Wojciech Zaremba. **Openai gym** (2016). DOI: 10.48550/ARXIV.1606.01540. URL: https://arxiv.org/abs/1606.01540 (see page 6).

[Cah11] Andy Cahill. **Catastrophic forgetting in reinforcement-learning environments**. MA thesis. University of Otago, 2011. URL: http://hdl.handle.net/10523/1765 (see page 34).

[CLH18] Jui-Hung Chang, Yin Chung Leung and Ren-Hung Hwang. **A survey and implementation on neural network visualization**. In: *2018 15th international symposium on pervasive systems, algorithms and networks (i-span)*. 2018, 107–112. DOI: 10.1109/I-SPAN.2018.00026 (see page 6).

[den15] Arnoud V. den Boer. **Dynamic pricing and learning: historical origins, current research, and new directions**. *Surveys in operations research and management science* 20:1 (2015), 1–18. ISSN: 1876-7354. DOI: https://doi.org/10.1016/j.sorms.2015.03.001. URL: https://www.sciencedirect.com/science/article/pii/S1876735415000021 (see page 5).

[Dre22] Leonard Dreeßen. **Pricing in the re-commerce domain: analysis of pricing strategies with an online market simulation**. Hasso-Plattner-Institute, 2022 (see pages 7, 33).

[EIT13] Jacqueline K Eastman, Rajesh Iyer and Stephanie P Thomas. **The impact of status consumption on shopping styles: an exploratory look at the millennial generation**. *Marketing management journal* 23:1 (2013), 57–73 (see pages 8, 43).

[FHM18] Scott Fujimoto, Herke van Hoof and David Meger. **Addressing function approximation error in actor-critic methods**. In: *Proceedings of the 35th international conference on machine learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, 1587–1596. URL: https://proceedings.mlr.press/v80/fujimoto18a.html (see page 14).

[GB22] Torsten J. Gerpott and Jan Berends. **Competitive pricing on online markets: a literature review**. *Journal of revenue and pricing management* (June 2022). ISSN: 1477-657X. DOI: 10.1057/s41272-022-00390-x. URL: https://doi.org/10.1057/s41272-022-00390-x (see page 5).

[Gee+19] Ruben van de Geer, Arnoud V. den Boer, Christopher Bayliss, Christine S. M. Currie, Andria Ellina, Malte Esders, Alwin Haensel, Xiao Lei, Kyle D. S. Maclean, Antonio Martinez-Sykora, Asbjørn Nilsen Riseth, Fredrik Ødegaard and Simos Zachariades. **Dynamic pricing and learning with competition: insights from the dynamic pricing challenge at the 2017 informs rm & pricing conference**. *Journal of revenue and pricing management* 18:3 (June 2019), 185–203. ISSN: 1477-657X. DOI: 10.1057/s41272-018-00164-4. URL: https://doi.org/10.1057/s41272-018-00164-4 (see page 5).

[Gro22] Jan Niklas Groeneveld. **A comparison of reinforcement learning algorithms for dynamic pricing in recommerce markets**. Hasso-Plattner-Institute, 2022 (see pages 3, 13).

[GWZ99] Chris Gaskett, David Wettergreen and Alexander Zelinsky. **Q-learning in continuous state and action spaces**. In: *Advanced topics in artificial intelligence*. Ed. by Norman Foo. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, 417–428. ISBN: 978-3-540-46695-6 (see page 13).

[Haa+18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel and Sergey Levine. **Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor**. *Corr* abs/1801.01290 (2018). arXiv: 1801.01290. URL: http://arxiv.org/abs/1801.01290 (see pages 14, 23).

[Hen+17] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup and David Meger. *Deep reinforcement learning that matters*. 2017. DOI: 10.48550/ARXIV.1709.06560. URL: https://arxiv.org/abs/1709.06560 (see pages 6, 34).

[Her22] Judith Herrmann. **Scalable learning in the cloud**. Hasso-Plattner-Institute, 2022 (see page 21).

[Hun07] J. D. Hunter. **Matplotlib: a 2d graphics environment**. *Computing in science & engineering* 9:3 (2007), 90–95. DOI: 10.1109/MCSE.2007.55 (see page 16).

[Ins20] Wuppertal Institut. *Reuse und secondhand in deutschland*. Accessed: 2022-06-21. 2020. URL: https://de.statista.com/statistik/daten/studie/1248873/umfrage/bevorzugter-kanal-fuer-den-verkauf-von-secondhand-produkten-in-deutschland/ (see page 1).

[Isl+17] Riashat Islam, Peter Henderson, Maziar Gomrokchi and Doina Precup. **Reproducibility of benchmarked deep reinforcement learning tasks for continuous control**. *Corr* abs/1708.04133 (2017). arXiv: 1708.04133. URL: http://arxiv.org/abs/1708.04133 (see page 34).

[Jor+20] Scott Jordan, Yash Chandak, Daniel Cohen, Mengxue Zhang and Philip Thomas. **Evaluating the performance of reinforcement learning algorithms**. In: *Proceedings of the 37th international conference on machine learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 13–18 Jul 2020, 4962–4973. URL: https://proceedings.mlr.press/v119/jordan20a.html (see page 6).

[KHG00] Jeffrey O. Kephart, James E. Hanson and Amy R. Greenwald. **Dynamic pricing by software agents**. *Computer networks* 32:6 (2000), 731–752. ISSN: 1389-1286. DOI: https://doi.org/10.1016/S1389-1286(00)00026-8. URL: https://www.sciencedirect.com/science/article/pii/S1389128600000268 (see page 5).

[KLM96] Leslie Pack Kaelbling, Michael L. Littman and Andrew W. Moore. **Reinforcement learning: a survey**. *Journal of artificial intelligence research* 4 (1996), 237–285. DOI: https://doi.org/10.1613/jair.301. URL: https://www.jair.org/index.php/jair/article/view/10166 (see page 13).

[KPM20] KPMG. *Wie äußert sich bei ihnen der fokus auf nachhaltige mode beim shopping?* Accessed: 2022-06-21. 2020. URL: https://de.statista.com/statistik/daten/studie/1179997/umfrage/umfrage-unter-verbrauchern-zu-nachhaltigem-modekauf-in-deutschland/ (see page 1).

[KRH17] Julian Kirchherr, Denise Reike and Marko Hekkert. **Conceptualizing the circular economy: an analysis of 114 definitions**. *Resources, conservation and recycling* 127 (2017), 221–232. ISSN: 0921-3449. DOI: https://doi.org/10.1016/j.resconrec.2017.09.005. URL: https://www.sciencedirect.com/science/article/pii/S0921344917302835 (see page 2).

[Lil+15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver and Daan Wierstra. **Continuous control with deep reinforcement learning**. *Arxiv preprint arxiv:1509.02971* (2015) (see page 14).

[Mni+16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver and Koray Kavukcuoglu. **Asynchronous methods for deep reinforcement learning**. *Corr* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783 (see page 14).

[MPD02] José del R. Millán, Daniele Posenato and Eric Dedieu. **Continuous-action q-learning**. *Machine learning* 49:2 (Nov. 2002), 247–265. ISSN: 1573-0565. DOI: 10.1023/A:1017988514716. URL: https://doi.org/10.1023/A:1017988514716 (see page 13).

[Mye97] Roger B Myerson. **Game theory: analysis of conflict**. Harvard university press, 1997, 1 (see page 12).

[Nar+05] Y. Narahari, C. V. L. Raju, K. Ravikumar and Sourabh Shah. **Dynamic pricing models for electronic business**. *Sadhana* 30:2 (Apr. 2005), 231–256. ISSN: 0973-7677. DOI: 10.1007/BF02706246. URL: https://doi.org/10.1007/BF02706246 (see page 11).

[Ove22] Stack Overflow. *Stack overflow annual developer survey*. Accessed: 2022-06-23. 2022. URL: https://insights.stackoverflow.com/survey (see page 5).

[PAI22] PAIR-Code. *What-if tool*. Accessed: 2022-06-18. 2022. URL: https://pair-code.github.io/what-if-tool/ (see page 16).

[Pas+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai and Soumith Chintala. **Pytorch: an imperative style, high-performance deep learning library**. *Corr* abs/1912.01703 (2019). arXiv: 1912.01703. URL: http://arxiv.org/abs/1912.01703 (see page 13).

[Raf+21] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus and Noah Dormann. **Stable-baselines3: reliable reinforcement learning implementations**. *Journal of machine learning research* (2021) (see page 14).

[RG08] Dominique Roux and Denis Guiot. **Measuring second-hand shopping motives, antecedents and consequences**. *Recherche et applications en marketing (english edition)* 23:4 (2008), 63–91. DOI: 10.1177/205157070802300404. eprint: https://doi.org/10.1177/205157070802300404. URL: https://doi.org/10.1177/205157070802300404 (see pages 33, 44).

[SB18] Richard S Sutton and Andrew G Barto. **Reinforcement learning: an introduction**. MIT press, 2018 (see page 14).

[Sch+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov. **Proximal policy optimization algorithms**. *Corr* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347 (see page 14).

[SK86] George B. Sprotles and Elizabeth L. Kendall. **A methodology for profiling consumers' decision-making styles**. *Journal of consumer affairs* 20:2 (1986), 267–279. DOI: https://doi.org/10.1111/j.1745-6606.1986.tb00382.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1745-6606.1986.tb00382.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1745-6606.1986.tb00382.x (see page 43).

[Ten20] TensorFlow. *Developing a tensorboard plugin*. Accessed: 2022-06-18. 2020. URL: https://github.com/tensorflow/tensorboard/blob/master/ADDING_A_PLUGIN.md (see page 16).

[TP19] Linda Lisa Maria Turunen and Essi Pöyry. **Shopping with the resale value in mind: a study on second-hand luxury consumers**. *International journal of consumer studies* 43:6 (2019), 549–556. DOI: https://doi.org/10.1111/ijcs.12539. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/ijcs.12539. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/ijcs.12539 (see page 33).

[Wex+20] James Wexler, Mahima Pushkarna, Tolga Bolukbasi, Martin Wattenberg, Fernanda Viégas and Jimbo Wilson. **The what-if tool: interactive probing of machine learning models**. *Ieee transactions on visualization and computer graphics* 26:1 (2020), 56–65. DOI: 10.1109/TVCG.2019.2934619 (see page 16).

[Won+18] Kanit Wongsuphasawat, Daniel Smilkov, James Wexler, Jimbo Wilson, Dandelion Mané, Doug Fritz, Dilip Krishnan, Fernanda B. Viégas and Martin Wattenberg. **Visualizing dataflow graphs of deep learning models in tensorflow**. *Ieee transactions on visualization and computer graphics* 24:1 (2018), 1–12. DOI: 10.1109/TVCG.2017.2744878 (see page 6).

List of Figures

A Appendix

A.1 Tables

Consumer Characteristic	Description
Perfectionistic, High-Quality Conscious	Consumer searches carefully and systematically for the very best quality in products
Brand Conscious, 'Price = Quality'	Consumer is oriented towards buying the more expensive, well-known brands
Novelty and Fashion Conscious	Consumers who like new and innovative products and gain excitement from seeking out new things
Recreational and Shopping Conscious	Consumer finds shopping a pleasant activity and enjoys shopping just for the fun of it
Price Conscious/ Value for the Money	Consumer with a particularly high consciousness of sale prices and lower prices in general
Impulsive/ Careless	Consumer who buys on the spur of the moment and appears unconcerned about how much he/she spends
Confused by Overchoice	Consumer perceiving too many brands and stores from which to choose and experiences information overload in the market
Habitual/ Brand Loyal	Consumer who repetitively chooses the same favorite brands and stores

Table A.1: Consumer Shopping Styles, from [EIT13], including information from [SK86].

I - Economic dimensions
1. ECO1 - Buying cheaper, spending less (anxiety expressed in regard to expenditure) 2. ECO2 - Paying fair prices 3. ECO3 - Allocative role of price (what is obtained for a particular budget) 4. ECO4 - Bargain hunting
II - Dimensions relating to the nature of the offering
5. OFF1 - Originality 6. OFF2 - Nostalgia 7. OFF3 - Congruence 8. OFF4 - Self-expression
III - Dimensions relating to the recreational aspects of second-hand channels
9. CIR1 - Social contact 10. CIR2 - Stimulation 11. CIR3 - Treasure hunting
IV - Power dimensions
12. PUIS1 - Smart shopping 13. PUIS2 - Power over the seller
V - 14. ETH - Ethical and ecological dimension
VI - 15 ANT-OST - Anti-ostentation dimension

Table A.2: 15 areas of motivation toward second-hand shopping, from [RG08] (descriptions omitted).

		state/in_circulation	state/in_storage	action/price_new	action/price_refurbished	action/rebuy_price	owner/throw_away	owner_rebuys	customer/purchase_new	customer/purchase_refurbished	customer/buy_nothing	profit/rebuy_cost	profit/storage_cost	profit/by_new	profit/by_refurbished	profit/all	profit/reward
Exampleprinter		X		X	X	X	X	X	X	X						X	
TensorBoard		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Live	Scatter	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
	Line	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
Agent	Density	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	Violin	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	Line	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Table A.3: All metrics recorded during the simulation and which monitoring tools visualise them.

A.2 Exemplary calculation of customer purchase probabilities

In this section, we will calculate the probability distribution for an exemplary market scenario[1]. We assume a Circular Economy, Duopoly market scenario with prices set by the two vendors as shown in Table A.4.

	P_{new}	P_{ref}
Vendor 0	7	4
Vendor 1	6	2

Table A.4: Prices set by the vendors in our exemplary market scenario.

This will result in the following preference ratios for new products, following the definition in Equation (3.1):

$$r_{0,new} := r_{new}(P_{0,new}) = \frac{10}{7} - e^{7-8} \approx 1.06 \tag{A.1}$$

$$r_{1,new} := r_{new}(P_{1,new}) = \frac{10}{6} - e^{6-8} \approx 1.53 \tag{A.2}$$

We can immediately see that customers prefer the new product offered by vendor 1 over the one offered by vendor 0, signified by the higher preference ration, which is due to the lower price of the product offered by vendor 1. The same calculation can now be done for the refurbished products following Equation (3.2):

$$r_{0,ref} := r_{ref}(P_{0,ref}) = \frac{5.5}{4} - e^{4-5} \approx 1.01 \tag{A.3}$$

$$r_{1,ref} := r_{ref}(P_{1,ref}) = \frac{5.5}{2} - e^{2-5} \approx 2.70 \tag{A.4}$$

It seems that a price of 2 for a refurbished item is a great deal. In order to draw samples from the *multinomial* distribution, we must now normalise our preference ratios using *softmax*. For this, we first calculate the sum over all preference ratios as defined in Equation (3.4) (including the 'nothingpreference' of Equation (3.3)):

$$S = e^1 + e^{1.06} + e^{1.53} + e^{1.01} + e^{2.70} \approx 27.85 \tag{A.5}$$

Using this sum, we can now calculate the purchase probabilities for each product using Equation (3.5), the results of which can be seen in Table A.5.

	π_{new}	π_{ref}	π_{not}
Vendor 0	0.10	0.10	0.10
Vendor 1	0.17	0.53	

Table A.5: Purchase probabilites created by applying the *softmax* function on the different preference ratios.

These probabilites sum to 1, as required, and can subsequently be used to draw samples from the multinomial distribution, which will be omitted here.

1 Refer to Section 3.2 for the explanation of customer behaviour in our framework.

A.3 Rule-based agents - Policies

```
def policy(market_state) -> tuple:
  products_in_storage = market_state.products_in_storage
  price_refurbished = 0
  price_new = market.production_price
  rebuy_price = 0

  if products_in_storage < market.max_storage / 15:
    # try to fill up the storage immediately
    price_refurbished = market.max_price * 6/10
    price_new += market.max_price * 6/10
    rebuy_price = price_refurbished - 1

  elif products_in_storage < market.max_storage / 10:
    # try to fill up the storage
    price_refurbished = market.max_price * 5/10
    price_new += market.max_price * 5/10
    rebuy_price = price_refurbished - 2

  elif products_in_storage < market.max_storage / 8:
    # storage content is ok
    price_refurbished = market.max_price * 4/10
    price_new += market.max_price * 4/10
    rebuy_price = price_refurbished // 2

  else:
    # storage too full
    # try to get rid of some refurbished products
    price_refurbished = market.max_price * 2/10
    price_new += market.max_price * 7/10
    rebuy_price = 0

  price_new = min(9, price_new)
  return (price_refurbished, price_new, rebuy_price)
```

Figure A.1: Policy implementation of the *RuleBasedCERebuyAgent*, simplified for readability.

```
def policy(market_state) -> tuple:
  own_storage = market_state.own_storage
  comp_refurbished, comp_new, comp_rebuy = get_competitor_prices()

  price_new = max(median(comp_new) - 1, market.production_price + 1)

  if own_storage < market.max_storage / 15:
    # try to fill up the storage immediately
    price_refurbished = max(comp_new + comp_refurbished)
    rebuy_price = price_new - 1

  else:
    # storage too full, try to get rid of some refurbished products
    rebuy_price = min(comp_rebuy) - market.max_price / 0.1
    price_refurbished = np.quantile(comp_refurbished, 0.25)

  return clamped_prices(price_refurbished, price_new, rebuy_price)
```

Figure A.2: Policy implementation of the *RuleBasedCERebuyAgentStorageMinimizer*, simplified for readability.

```
  def policy(market_state) -> tuple:
    own_storage = market_state.own_storage
    comp_refurbished, comp_new, comp_rebuy = get_competitor_prices()
    price_new = max(min(comp_new) - 1, market.production_price + 1)

    if own_storage < market.max_storage / 15:
      # try to fill up the storage immediately
      price_refurbished = min(comp_refurbished) + 1
      rebuy_price = max(min(comp_rebuy) + 1, 2)
    elif own_storage < market.max_storage / 8:
      # storage content is ok
      rebuy_price = max(min(comp_rebuy) - 1, 0.25)
      price_refurbished = max(min(comp_refurbished) - 1, rebuy_price + 1)
    else:
      # storage too full, try to get rid of some refurbished products
      rebuy_price = max(min(comp_rebuy) - 2, 0)
      price_refurbished = max(np.quantile(comp_refurbished, 0.75) - 2, rebuy_price + 1)

    return clamped_prices(price_refurbished, price_new, rebuy_price)
```

Figure A.3: Policy implementation of the *RuleBasedCERebuyAgentCompetitive*, simplified for readability.

A.4 SAC-Duopoly experiment

A.4.1 Configuration files

```
{
  "task": "training",
  "marketplace": "CircularEconomyRebuyPriceDuopoly",
  "agents": [
    {
      "name": "StableBaselinesSAC",
      "agent_class": "StableBaselinesSAC",
      "argument": ""
    },
    {
      "name": "RuleBasedCERebuyAgentCompetitive",
      "agent_class": "RuleBasedCERebuyAgentCompetitive",
      "argument": ""
    }
  ]
}
```

Figure A.4: The `environment_config.json` of the SAC-Duopoly experiment, simplified for readability.

```
{
  "max_storage": 100,
  "episode_length": 50,
  "max_price": 10,
  "max_quality": 50,
  "number_of_customers": 20,
  "production_price": 3,
  "storage_cost_per_product": 0.1,
  "opposite_own_state_visibility": true,
  "common_state_visibility": true,
  "rewards_mixed_profit_and_difference": false,
}
```

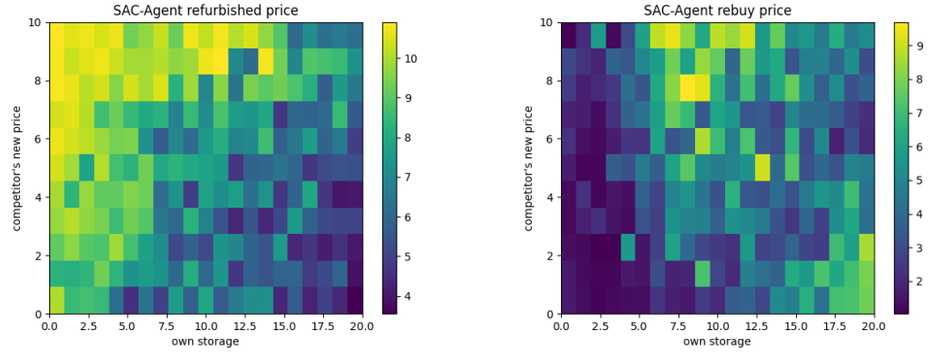
Figure A.5: The `market_config.json` of the SAC-Duopoly experiment.

```
{
  "learning_rate": 3e-4,
  "buffer_size": 1000000,
  "learning_starts": 100,
  "batch_size": 256,
  "tau": 0.005,
  "gamme": 0.99,
  "ent_coef": "auto",
}
```

Figure A.6: The configuration file for the SAC-Agent of the SAC-Duopoly experiment.

A.4.2 Additional diagrams - Policyanalyser

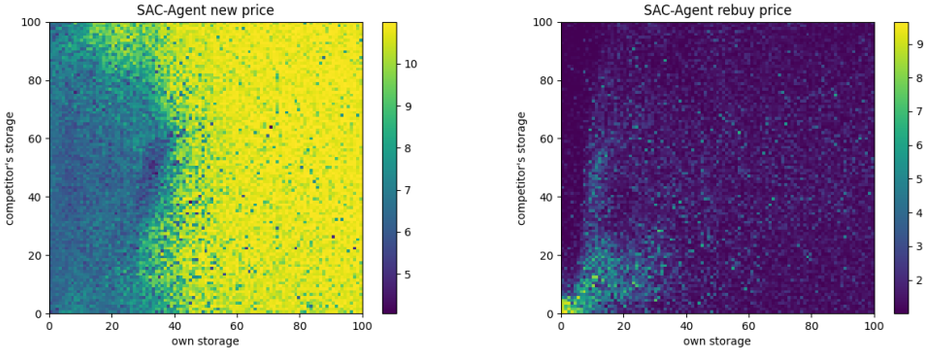
Figure A.7 and Figure A.8 show additional results of a Policyanalyser session run on the trained SAC-Agent. In Figure A.7 (a) we can see that the lower the agent's storage and the higher the competitor's new price, the higher the agent will set the price for its refurbished products. This is the result of both the agent seeing that it can increase prices and still be cheaper than its opponent, and the agent lowering prices to sell more refurbished products if storage is full to reduce storage costs. Figure A.7 (b) shows that rebuy prices are low if storage is low, but the competitor's new prices also slightly, but inconsistently, influence rebuy prices.



(a) Refurbished prices **(b)** Rebuy prices

Figure A.7: Prices set by the trained SAC-Agent, depending on the competitor's new price and the agent's own storage.

Figure A.8 (a) shows that new prices rise the more items are in the agent's storage, this is to incentivise customers to rather buy refurbished products, which will decrease inventory and thereby storage costs. Competitor's storage seems to have close to no effect on the agent's new price. Figure A.8 (b) shows that rebuy prices are only high of inventory is very low, which is the result in the agent trying to always have at least some products in storage, as otherwise it would not be able to sell refurbished products. In anticipation of low rebuy prices by its competitor, the agent also sets low rebuy prices of competitor storage is low.



(a) New prices **(b)** Rebuy prices

Figure A.8: Prices set by the trained SAC-Agent, depending on both the competitor's and the agent's storage.

A.5 PPO-Oligopoly experiment

A.5.1 Configuration files

The `market_config.json` is the same as the one for the SAC-Duopoly experiment and can be found in Figure A.5.

```
{
  "task": "training",
  "marketplace": "CircularEconomyRebuyPriceOligopoly",
  "agents": [
    {
      "name": "StableBaselinesPPO",
      "agent_class": "StableBaselinesPPO",
      "argument": ""
    },
    {
      "name": "RuleBasedCERebuyAgentCompetitive",
      "agent_class": "RuleBasedCERebuyAgentCompetitive",
      "argument": ""
    },
    {
      "name": "RuleBasedCERebuyAgent",
      "agent_class": "RuleBasedCERebuyAgent",
      "argument": ""
    },
    {
      "name": "FixedPriceCERebuyAgent",
      "agent_class": "FixedPriceCERebuyAgent",
      "argument": [4, 7, 2]
    },
    {
      "name": "RuleBasedCERebuyAgentStorageMinimizer",
      "agent_class": "RuleBasedCERebuyAgentStorageMinimizer",
      "argument": ""
    }
  ]
}
```

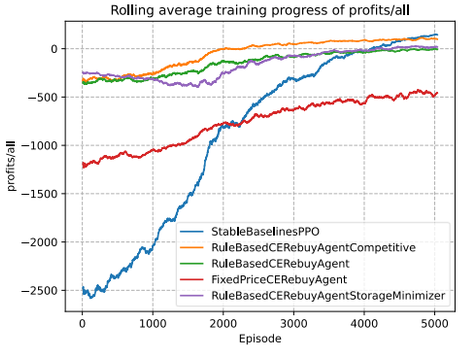
Figure A.9: The `environment_config.json` of the PPO-Oligopoly experiment, simplified for readability.

```
{
  "learning_rate": 3e-4,
  "n_steps": 2048,
  "batch_size": 64,
  "n_epochs": 10,
  "gamma": 0.99,
  "clip_range": 0.2
}
```

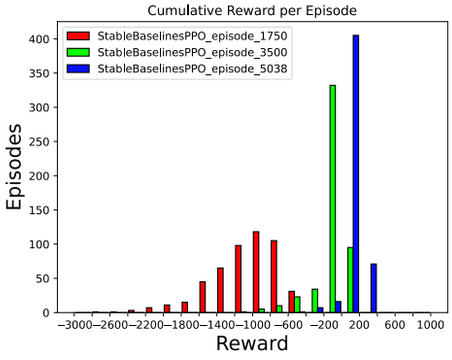
Figure A.10: The configuration file for the PPO-Agent of the PPO-Oligopoly experiment.

A.5.2 Diagrams

Some of the more interesting diagrams created during the training and subsequent Live-monitoring of the PPO-Oligopoly experiment are shown here, without in-depth interpretation. The agent was trained for a total of 5,000 episodes. Following the standard Oligopoly setup, all vendors played at the same time, setting prices after one another within each step.

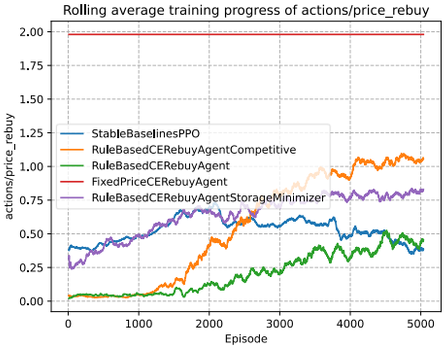


(a) Profits per episode achieved during the training run

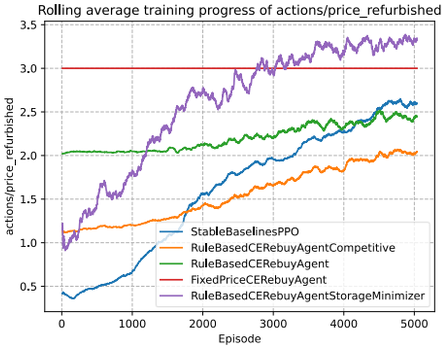


(b) Cumulative rewards per training stage per interval, recorded during the Agent-monitoring session

Figure A.11: The PPO-Agent took significantly longer than the SAC-Agent (Figure 6.1) to reach its maximum possible profit (Figure A.11 (a)) and the model that was trained longest outperforms the other two (Figure A.11 (b)), as was to be expected following the steady increase in profits during training.



(a) Rebuy prices



(b) Prices for refurbished products

Figure A.12: While rebuy prices were consistently at or below 1 (excluding the *FixedPriceAgent*), prices for refurbished products rose consistently, with the *RuleBasedCERebuyAgentStorageMinimizer* leading the price run.

(a) Number of purchases of refurbished products **(b)** Storage costs per episode

Figure A.13: Except in the case of the *FixedPriceAgent*, a higher number of sales of refurbished products was always followed by a similar decrease in storage costs, meaning that the number of bought back products likely stayed on a steady level over the course of training. This is confirmed by Figure A.14 (a).

(a) Number of products bought back by each vendor **(b)** Number of customers that bought no product

Figure A.14: The number of products bought back by vendors (except for the *FixedPriceAgent*) stayed similar over the whole course of training. With an increase in rebuy-prices (Figure A.12 (a)), the *FixedPriceAgent* lost some of its rebuys to the other vendors. Following an overall increase in prices (see e.g. Figure A.12 (b)), more and more customers chose to buy none of the advertised products.

Declaration of Authorship

I hereby declare that this thesis is my own unaided work. All direct or indirect sources used are acknowledged as references.

Potsdam, 27th June 2022

Nikkel Mollenhauer