Bachelor Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Data-Driven Automated Dynamic Pricing for E-commerce

Jiří Moravčík

Supervisor: Ing. Jan Mrkos
Study program: Open Informatics
Specialisation: Artificial Intelligence and Computer Science
May 2021

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Moravčík Jiří**	Personal ID number:	**483741**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Data-Driven Automated Dynamic Pricing for E-commerce

Bachelor’s thesis title in Czech:

Automatizovaná dynamická cenotvorba pro e-shop na základě dat

Guidelines:

The goal of this thesis is to propose, develop and test a data-driven dynamic pricing solution for an e-commerce platform. To this end, student is to achieve the following objectives:
1. Collect, and clean data generated by the e-commerce platform into a dataset. Analyze the dataset, propose suitable metrics and features base on the dataset.
2. Perform a survey of methods used for dynamic pricing in e-commerce. Focus on models based on Markov decision process and methods such as offline reinforcement learning or imitation learning. Evaluate drawbacks and benefits of the methods regarding the available data. Propose a method for implementation.
3. Implement a dynamic pricing solution. Design data features to be used with the solution and evaluation metrics for evaluating the performance of the solution.
4. Evaluate the performance of the method. Test on testing data.

Bibliography / sources:

[1] Russell, Stuart J. and Norvig, Peter - Artificial Intelligence: A Modern Approach (2nd Edition) – 2002
[2] Liu Jiaxi et al. - Dynamic Pricing on E-commerce Platform with Deep Reinforcement Learning - 2019
[3] Osa Takayuki et al. - An Algorithmic Perspective on Imitation Learning - 2018
[4] Sutton, Richard S. and Barto, Andrew G.. Reinforcement Learning: An Introduction. Second: The MIT Press, 2018
[5] Arnoud V den Boer. Dynamic pricing and learning: historical origins, current research, and new directions. Surveys in operations research and management science, 20(1):1–18, 2015

Name and workplace of bachelor’s thesis supervisor:

Ing. Jan Mrkos, Artificial Intelligence Center, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **08.01.2021** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **30.09.2022**

Ing. Jan Mrkos	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I would like to thank my supervisor Ing. Jan Mrkos for guidance, insights and help with this work.

Furthermore, I would like to thank everyone that supported me while I worked on this thesis.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 21. May 2021

Abstract

In this thesis, we propose a novel approach to dynamic pricing in the setting of e-commerce, specifically fashion retail. For the first time, we attempt to use the methods of Offline Reinforcement Learning and Imitation Learning in the domain of dynamic pricing. We use two algorithms, namely, the Conservative Q-learning and Behavioral Cloning.

We formalize the pricing problem as a Markov Decision Process, then we apply the proposed algorithms to a real-world dataset from a small e-commerce business. The results show that our methods are able to achieve good results on par with human experts. We carry out two separate evaluations. In the first one, a human expert grades the pricing recommendations of the best method with an average grade 1.31 out of 5 (1 being the best possible score). The second evaluation shows that all our proposed methods outperform the static baseline method by a minimum margin of 27%. We perform this simulated evaluation via a custom simulator implemented in OpenAI's Gym. For the expert evaluation, we develop a custom methodology based on a web interface.

Keywords: dynamic pricing, reinforcement learning, e-commerce

Supervisor: Ing. Jan Mrkos

Abstrakt

V této práci navrhujeme nový přístup k dynamické cenotvorbě v prostředí e-commerce, konkrétně internetového obchodu s módou. Poprvé se pokusíme využít metod offline zpětnovazebního učení a napodobovacího učení v oblasti dynamické cenotvorby. Používáme dva algoritmy, a to konzervativní Q-učení a klonování chování.

Formalizujeme problém cenotvorby jako Markovův rozhodovací proces, pak aplikujeme navrhované algoritmy na skutečnou datovou sadu z malého internetového obchodu. Výsledky ukazují, že naše metody jsou schopné dosáhnout dobrých výsledků srovnatelných s lidskými odborníky. Provádíme dvě samostatné evaluace. V první z nich lidský expert hodnotí cenové doporučení nejlepší metody s průměrnou známkou 1,31 z 5 (1 je nejlepší možné skóre). Druhá evaluace ukazuje, že všechny naše navrhované metody překonávají základní statickou metodu minimálně o 27 %. Tuto simulovanou evaluaci provádíme pomocí vlastního simulátoru implementovaného ve frameworku OpenAI Gym. Pro exeprtní evaluaci vytváříme vlastní metodiku založenou na webovém rozhraní.

Klíčová slova: dynamická cenotvorba, zpětnovazební učení, e-commerce

Překlad názvu: Automatizovaná dynamická cenotvorba pro e-shop na základě dat

Contents

Figures

Tables

Chapter 1

Introduction

1.1 General Overview

Dynamic pricing is an area of revenue optimization. Various scientific fields, including operations research, management science, marketing, econometrics, and computer science have extensively studied dynamic pricing. This work focuses on the usages of machine learning in the field of dynamic pricing, mainly focusing on the usage of Offline Reinforcement Learning (RL). For a comprehensive overview, please refer to Chapter 2.

Businesses have always tried to set the prices of their goods and services in a way that would either maximize their revenue or customer satisfaction. Determining such prices is a complex task and this is where dynamic pricing comes into play. The usage ranges from hospitality (higher prices during peaks in the season), transport (airplane tickets, surge pricing in ride-sharing services), to retail (e-commerce product pricing). The task requires that the seller has extensive knowledge of their own costs and supply elasticity but also the knowledge of their customer's needs and possible future demand.

With the rise of big data and general improvements in data collection, companies have better opportunities to predict their customer's behavior based on the collected data. Another relevant factor is the competition, nowadays price comparison services enable customers to see all prices in one place. If the seller wants to gain an advantage, the seller should take the competition into account.

The idea of having a powerful autonomous algorithm behind one's business is very appealing and nowadays, many businesses are trying to adopt such approaches to pricing. The benefits include decreased labor costs, increased profit, and other possibilities of optimizing the seller's resources.

1.2 Dynamic Pricing on an E-commerce Platform

One suitable domain for dynamic pricing is an e-commerce platform. E-commerce solutions are usually implemented as websites offering various products, the differences compared to normal stores include the ability to choose and purchase products without actually needing to go to the physical store and having the products delivered directly to the customer's home (although not always the case). Because of these features, such businesses saw a large increase in popularity, the outbreak of COVID-19 virus also helped this phenomenon.

Nowadays, e-commerce businesses try to maximize their revenue in a highly competitive market where buyers are able to easily compare products and their prices. At the same time, modern analytic and big data tools allow sellers to collect large amounts of data about their customers' behavior.

Dynamic pricing based on collected data can help businesses to mitigate risks connected with a very dynamic environment of online retail like demand fluctuation and price battles between competitors. For the sake of completeness, we refer the reader to a complete overview of dynamic pricing models used in electronic business [33].

Fashion E-commerce Dynamic Pricing Domain Description

The domain considered in this work is a small e-commerce business. The main focus of the business are cosmetics, clothes, and accessories for men. A large part of the business consists of fashion, the owners have their own clothes brand, and one of such products are ready-to-wear shirts, which are studied in this work. We have data from over 3 years of sales including customer traffic like unique page views and average time spent on the page (for more information about the data, please refer to Chapter 3).

We will now formalize the environment for dynamic pricing in e-commerce for our specific needs. Please refer to Section 4.1 for an exact algorithmic formulation. We consider all shirts to be priced in an aggregated manner, because the amount of data required to train dynamic pricing algorithms is large (for more information about the algorithms, please refer to Chapter 4).

The pricing update process will be performed on a monthly basis. The data available does not allow shorter update periods due to its small size and unfriendly data access. The data features used in pricing consist of: the product's current price, the product's stock price, unique page visits for a given product, or the number of stock available. We describe the data in Section 3.2.

For completeness, we provide an exact specification of the products. The ready-to-wear shirts, whose prices we will be trying to optimize are considered seasonal from the business perspective. We classify the pricing problem as that of finite-stock perishable products for the following reasons:

- finite-stock — the seasonal shirts are manufactured for a given season and never get restocked,
- perishable — the business objective is to sell the shirts in a horizon of one year, because the warehouse needs space for the next seasonal shirts, thus we optimize for a finite selling horizon.

Main Design Decisions

We proceed to define three important questions that have to be answered when trying to employ a dynamic pricing solution. The first important decision we have to make is to define the price change interval. We may decide to keep a strictly regular change interval such as day, week, or month. The other option is doing irregular price changes. In that case, we can base them on specific events important for us, such as a surge in customer traffic or stocking of new products. Due to the nature of our dataset, we decided to use a regular change interval of one month.

The second decision is whether we should price each product separately or not. Here, we consider a product to be the most granular unit that a given business can sell, e.g., blue checked shirts. If we choose to price each product separately, we may not have enough historical data to create good pricing decisions. On the other hand, if we decide to price several products the same, we may run into issues with data bias. If we price several similar products (similar products could be shirts with different colors) the same, we are creating a price approximation based on multiple products that can lead to errors.

These errors clearly depend on the size of the product aggregation. In an ideal world, we would have enough data for every product, but that happens rarely. Therefore, we should carefully examine our data and perform an extensive study to make an informed decision on which products to aggregate. In our case, we choose to aggregate the priced products due to the small size of our dataset (see Section 3.2 for more information).

The last and the most important decision is evaluation. We have to choose how to evaluate the performance of our dynamic pricing solution. And this choice is difficult. The market is a highly complex environment, therefore we cannot accurately simulate the dynamics of such system. On the other hand, a simple simulation is useful for the development phase to get a rough idea of the pricing solution's performance. We propose a custom simulator for this purpose in Section 5.2.

A natural idea for evaluation is deployment into the real market. In our case, this choice is not feasible for two reasons. The first reason is the danger of capital loss in case of wrong choices made by the pricing algorithm. Prices set too high would mean that products will not be sold and occupy the warehouse. Overly low prices would lose revenue to customers that are willing to pay more. Weird pricing patterns could damage the seller's reputation.

The second reason against the feasibility of real market deployment is the length of period needed to collect data. In our case, we would need a minimum of twelve months to collect a dataset for evaluation. Due to the nature of this work, this is not possible. One possible way out of this problem is an evaluation performed by domain experts (see Section 6.1). For more information about the experiments and evaluation of this work, please refer to Chapter 6.

Methods

Having determined the domain and the kind of dynamic pricing we would like to use, we are additionally constrained by the real-world dataset.

Usually, Reinforcement Learning (RL) assumes access to a simulator, which we do not have. Recently, a new subfield of RL called Offline RL emerged, Offline RL assumes no access to a simulator and learns exclusively from prior offline data.

It turns out that this work is actually the first attempt to approach dynamic pricing using methods of Offline RL. Specifically, we use the algorithm Conservative Q-learning (CQL), and our evaluation results show that this approach is feasible and achieves great results (see Chapter 6 for evaluation). We also use the methods of Imitation Learning (IL), with the specific algorithm being Behavioral Cloning (BC), which is able to learn from offline data only.

Outline of the Thesis

We introduce the general concept of dynamic pricing and discuss its usages in Chapter 1, this work successfully uses the methods of Offline RL for the first time in the context of dynamic pricing.

Researchers have studied dynamic pricing in many scientific fields, this work approaches the dynamic pricing problem with the methods of RL and IL. We provide a literature review on these topics in Chapter 2. The data used for training and evaluation is an important part of every machine learning project, we describe the data retrieval pipeline and the data itself in Chapter 3.

We need to pair the data with an algorithm to produce pricing decisions, this work uses the algorithms Conservative Q-learning (CQL) and Behavioral Cloning (BC). Chapter 4 defines the algorithms CQL and BC with the necessary context. The underlying model of the environment used in the algorithms is a Markov Decision Process (MDP), we discuss the model with respect to our domain in Chapter 5.

We put the data, algorithms, and the model together in Chapter 6, where we perform the evaluation of this work, which shows that both CQL and BC outperform baseline methods and are able to train on a small real-world dataset. Specifically, we provide two methods of evaluation: domain expert evaluation (Section 6.1), and simulation (Section 6.2) We conclude the work with a high-level evaluation of the results, discussion of the strong and weak sides of this work, and mention some ideas for future work in Chapter 7.

Chapter 2

Literature Review

In this chapter, we present an overview of the literature on the topics of dynamic pricing and RL. We look at dynamic pricing in Section 2.1, we provide an overview of RL in Section 2.2.

2.1 Dynamic Pricing

The topic of dynamic pricing has been studied in many scientific fields. While the scope of this work does not allow us to review the whole field of dynamic pricing, we will look into the area of dynamic pricing concerned with machine learning. For a comprehensive overview of dynamic pricing with a broad scope, we refer the reader to [6].

The first attempts to use RL for dynamic pricing date to 1998, when researchers from IBM Watson proposed a method [44] for multi-agent dynamic pricing based on methods of RL, specifically Dynamic Programming. The same researchers then followed up with the usage of neural networks [43] (1999) and regression trees [40] (2002) paired with Q-learning. [7] (2001) created a platform for analyzing dynamic pricing called "Learning Curve Simulator" that uses different demand curves to simulate the price elasticity of demand. Futuristic expectations of a global economy merged with the internet are envisioned in [20] (2000), where the authors coin out a term *pricebot* — an intelligent agent based on Machine Learning methods (one example given is Q-learning) that competes in a multi-agent setting while trying to optimally price goods and services.

A broad overview on dynamic pricing for electronic businesses is provided in [33], the authors look at different models for dynamic pricing and then expand on the usage of RL in dynamic pricing with a simple simulation using a Poisson process as a customer arrival model. [46] uses RL in multi-agent setting for dynamic pricing in a Grid market environment. The algorithm used in this work is based on Policy-gradient theorem introduced in [42]. The authors claim that this is the first usage of gradient-based methods in the setting of dynamic pricing. Policy-gradient theorem and the usage of gradient descent in the optimization of neural networks is a pivotal topic of Deep Reinforcement Learning (see Section 4.4).

Simulation of pricing environments is a difficult task and for the purposes of modelling, we may choose to create a simple simulation to get a rough idea of the algorithm's performance. One good example of such an approach is [19], the proposed simulation uses a Poisson process for arrival rate and a uniform distribution as the acceptable price for a given product, the inventory of products is finite. The setting considers two homogeneous sellers. Authors then proceed to test their algorithms based on RL (specifically Q-learning) in the mentioned simulation. We draw inspiration from [19] and introduce our own simulation in Section 5.2.

[36] proposes a simple simulation of a single-seller market with customer arrival rates approximated via a Poisson process. The authors consider a discrete set of actions and solve the problem using Q-learning computed via the techniques of dynamic programming. This simulation is simple and interpretable, but the discrete set of actions renders it useless in domains with continuous action space.

The follow-up research from the same authors [35] considers a monopolistic retail market with customer segmentation. Customers are segmented into two groups, *captives* and *shoppers*. Captives are considered as loyal buyers with a higher acceptable price. On the other hand, shoppers are more price sensitive and get attracted by sales, promotions, etc. The algorithm used is Q-learning. The simulation used to evaluate uses a Poisson process for customer arrival rate and the acceptable price is modelled using a uniform distribution. While the base part of the simulation proposed by [35] is the same as [19], the addition of so-called captives and shoppers is not useful in the domain of fashion retail. [37] looks at the problem of interdependent products in a finite selling horizon and solves this task with a Temporal Difference Q-learning approach. Considering product interdependence is an interesting idea, but our data does not enable us to do so.

Researchers have looked at the problem of selling a given amount of stock in a finite horizon in the setting of dynamic pricing using RL [4] uses Q-learning paired with a predecessor of today's deep learning called "self-organizing map" or "map neural network". The simulation used to evaluate this method is using a Poisson process to model arrival rates and uses discrete coefficients for the uncertainty of demand. A follow-up work [5] studies a Q-learning approach based on real-time demand learning. The simulation here is a bit different. The author assumes that the arrival rate follows a gamma distribution.

The domain of this work is fashion retail. We cite two works that performed analytics and optimization of pricing for a fashion retailer. The first work [3] concerns itself with clearance pricing in the fast fashion retailer Zara. The authors perform a field experiment and establish a new process of pricing products during clearance sales. They report an increase of revenue by about 6%. The methodology uses sales data from past seasons to estimate the demand.

The latter [8] discusses demand forecasting for the fashion retailer Rue La La. The analysis is concerned with products that have never been sold before, specifically with their pricing and demand prediction. The authors report an improvement of the revenues by about 9.7%. The method used is based on Integer Linear Programming. While these works concern itself with fashion retail, they do not use the methods of RL to solve the dynamic pricing problem. This work attempts to use RL, specifically its offline variants, as a method to solve the problem of dynamic pricing in fashion retail.

The main paper used as an inspiration for this work is [29]. The authors present a formalization of the pricing environment using an MDP. We provide a similar one in Chapter 5. The methodology in [29] uses Deep Reinforcement Learning (DRL), for more info, see Section 4.3. Specifically, their dynamic pricing framework uses the algorithms Deep Q-learning from Demonstrations (DQfD)[17] and Deep Deterministic Policy Gradient from Demonstrations (DDPGfD)[45]. Their experiments and evaluations are performed on the website tmall.com based on the data from Alibaba. Field experiments show that dynamic pricing outperforms manual pricing.

2.2 Reinforcement Learning

RL has been a popular research topic in many scientific disciplines for several decades. The generality of RL makes it a viable research topic in many areas, such as operations research, game theory, multi-agent systems, etc. With this in mind, we have to declare that the scope of this work is not able to include a review of the whole field of RL. We refer to [41] for a review of the history of the field and its classical algorithms. We follow up with a more detailed review of DRL, and then we review its offline variants. At last, we review IL.

2.2.1 Deep Reinforcement Learning

Usage of deep neural networks as function approximators has brought large success in RL, [31], [32] achieved superhuman performance on several Atari games, using only pixels and game scores as input. The famous Nature journal published the article [32]. This accelerated the interest in the research areas of DRL. The mentioned papers present an algorithm called Deep Q-network (DQN), that uses a Replay Buffer. The usage of a Replay Buffer is important in Offline RL (see Section 4.5), where we prefill a Replay Buffer with the offline dataset and perform training without adding any more data to it.

Researchers have looked at the combination of actor-critic methods (see Section 4.4) and deep neural networks (see Section 4.3.1). The algorithm Deep Deterministic Policy Gradient (DDPG)[27] solved the same tasks as DQN but used a factor of 20 fewer steps. In addition to faster learning, DDPG works with continuous action spaces, unlike DQN, which works with discrete action spaces. The Asynchronous Advantage Actor-Critic (A3C)[30] is a DRL algorithm that does not use the Replay Buffer, yet it achieves good results. The main idea behind A3C is the parallelization of learning.

An important algorithm for this work is the Soft Actor-Critic (SAC) algorithm, because we use it as the base for the CQL algorithm. We use CQL to solve the dynamic pricing problem. SAC uses entropy regularization [14] and Clipped Double Q-learning [11], the algorithm itself was first presented in 2018 [15], but nowadays, there is a modern version that is simplified (does not learn a value function in addition to the Q-functions) [16]. We describe this modern version with the necessary context in Section 4.4.

Notable predecessors to the family of Offline RL algorithms are the algorithms DQfD[17] and DDPGfD[45]. They are based on DQN and DDPG algorithms. The difference is the introduction of a pretraining phase. We store the *demonstrations* in the Replay Buffer when we initialize the algorithm. We consider the demonstrations to be a static dataset of previously collected data. The agent pretrains itself using these demonstrations. While this accelerates learning when we switch to online learning, the agent trained only in the pretraining (offline) phase does not achieve reasonable results. See Section 4.2 for more information about online and offline learning.

We review algorithms that can achieve good results only using offline data in the next section 2.2.2.

2.2.2 Offline Reinforcement Learning

The modern era of Offline RL began with the Batch-Constrained Deep Q-learning (BCQ)[10] algorithm, the main idea of the algorithm is to learn from a fixed dataset that was collected in the past, and does not change. The algorithm uses a Variational Auto-Encoder (VAE)[22] as the generative model for perturbations that are used to constrain the data that was sampled from the Replay Buffer. The motivation behind this is the success of supervised algorithms (e.g., computer vision) learned on a large static dataset. Offline RL is sometimes called batch RL[25].

While this area of RL is quite new, researches have already made some progress. [23] presents the Bootstrapping Error Accumulation Reduction (BEAR) algorithm, the authors claim that it is less restrictive than BCQ, and it achieves superior results. The authors of BEAR have also released an overview over the Offline RL research [26]. The same team authored a project called *D4RL*[9], "Datasets for Deep Data-Driven Reinforcement Learning" based on OpenAI's Gym[2] environments using the MuJoCo physics simulation.

The algorithm used in this work, CQL[24] is a follow-up of BEAR. CQL achieves state-of-the-art results. Moreover, CQL is relatively simple to implement on top of other algorithms like SAC (see Section 4.4). For a rigorous algorithm description, see Section 4.6.

Even though there has been a lot of research in the area of dynamic pricing using RL, nobody has yet tried to use new algorithms that can train deep neural networks completely from offline data. This work's contribution is the first usage of Offline RL in the setting of dynamic pricing.

2.2.3 Imitation Learning

The topic of IL is an honorable mention in this review due to the fact that we use the BC algorithm. IL learns a policy based on expert behavior, i.e., IL tries to mimic some demonstrations in a dataset. We refer to [34] for an in-depth algorithmic overview of IL.

We use the RL toolkit Garage[12], which has a BC implementation based on [18]. For more information, see Section 4.7.

Chapter 3

Data Retrieval and Description

We use real-world data from a small e-commerce business to set up our pricing model. In this chapter, we describe the business domain, the data collection pipeline, and we describe and analyze the collected data.

The business is mainly focused on cosmetics, clothes, and accessories for men, offering a wide range of clothes, e.g., shirts, t-shirts, sweaters, boots, and jackets. A part of the clothes belong to a brand created by the business owners, these clothes are exclusive to the business that provided us with the data. No other business sells them, that is why we do not consider competitiveness as a factor in this work.

The primary market of the business consists of Czechia and Slovakia, but some other regions in Central Europe such as Poland or Hungary are also targeted with their own regional URLs. There is also one generic domain for customers from the whole of Europe, but it is the minority of market focus.

We narrow our focus to the seasonal shirts of the business owners' brand. Currently, the business performs a static discount procedure for the seasonal shirts. There are always two seasons in a year, summer and winter season. The business discounts the unsold shirts from the previous seasonal collection before introducing new shirts for the next season. The current procedure just makes two fixed discounts, the first one is 20% on all seasonal products and the second one is 40%, then the second one stays active until the products get sold out. This is the area in which we are trying to achieve an improvement, mainly by introducing a dynamic pricing method that can leverage historical data.

We describe the methods and challenges of data retrieval and cleaning in Section 3.1. The description of the data itself follows in Section 3.2.

3.1 Data Retrieval

In this section, we describe the data retrieval from external sources, data parsing, cleaning, and merging.

Starting with data retrieval, we retrieve the data from two separate sources:

- The e-commerce administration interface
- Google Analytics (GA)

The first source, the e-commerce administration interface, is a third party solution. This interface does not allow machine-friendly data export. Data mining is a common issue in many machine learning applications and our problem is no different. We choose to extract the data from raw HTML tables that are used to show statistics to the users of the administration interface. Currently, there is no other way to extract the required data from the interface.

This approach has one major pitfall: The data retrieved from the HTML tables is chunked into monthly periods. That is why it is not possible to have better time granularity like weeks or days. The absence of a better time granularity in the data does not enable us to examine details like sales on workdays vs weekend, etc. A Julia script performs the data extraction and parsing directly from the raw HTML tables. We use the extensions Gumbo and Cascadia for HTML parsing and extensions DataFrames and CSV for data merging.

We perform data cleaning and sanitizing on the data retrieved from the administration interface. Specifically, we convert dates and numbers into machine-friendly formats and redistribute stock data from one table row to others to achieve machine readability.

The second source is the GA platform API. This API is highly configurable and provides a reliable way of tracking customer traffic. The data we obtain from GA is split into monthly periods. While the GA API allows shorter time periods, we were limited by the data from the e-commerce administration interface. The data from GA serves as a complement to these data in terms of customer traffic. We obtain the actual data from the GA API via a Python script using the extension Google API Client. We also use the Python extension Pandas for data manipulation in this script.

Merging of the data from the administration interface and GA results in the final dataset used for training (see Chapter 6 for evaluation). The merging key is the product's URL. We store the dataset as a Comma-separated values (CSV) file. We provide an illustration of the whole data pipeline in Figure 3.1.

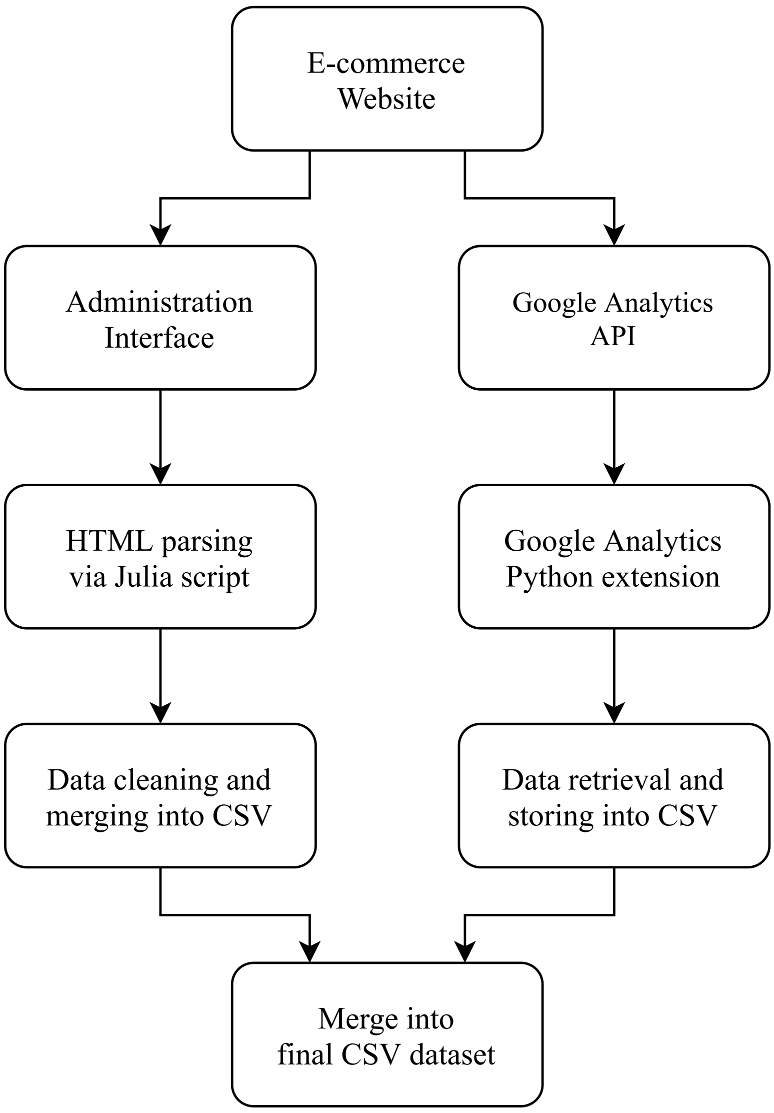

E-commerce Website
Administration Interface
Google Analytics API
HTML parsing via Julia script
Google Analytics Python extension
Data cleaning and merging into CSV
Data retrieval and storing into CSV
Merge into final CSV dataset

Figure 3.1: Diagram of the data retrieval pipeline

3.2 Data Description

In this section, we describe the data we retrieved via the pipeline described in the previous section.

We ran the pipeline on the records with their dates ranging from November 2017 to February 2021. We tried to obtain the largest training dataset possible, we ended up with a CSV file with 1902 rows. Each row represents a month of data for one variant of a shirt.

It is difficult to determine whether the size of 1902 rows is enough, but we managed to stabilize the learning and achieve good results on this dataset, for more information, see Chapter 6. It is important to note that we could not obtain a larger dataset. The data we have is the maximum that could be retrieved from the administration interface.

We ran the same data extraction pipeline on newer records, specifically those from March and April 2021. This CSV file contains 45 rows. We use this smaller dataset for evaluation in Section 6.1.

The CSV file of the dataset contains 8 columns that are used as dimensions of the state space (see Chapter 5 for more information about the state space). The columns concerned with price, stock, and total earned come from the e-commerce administration interface. The other columns related to website traffic come from GA. We provide a description of the relevant data columns in Table 3.1, the reader should be aware that the metrics are provided with respect to the given month.

We present the basic metrics of the data columns for the training set in Table 3.2. Furthermore, we emphasize that the dataset comes from a real-world e-commerce store, so we cannot assume a specific distribution. We also provide the same metrics for the evaluation dataset in Table 3.3. However, we look at the histogram of sell prices in Figure 3.2, this provides us with some insights about the data distribution. We provide the number of sales for each month in Figure 3.3.

Metric	Description (with respect to the given month)
Sale price	The sale price of the product
Stock price	The price the product was bought for by the business
Stock	The amount of products in the warehouse
Sold	The amount of sold products
Total Earned	The total revenue for all sold products
Unique page views	Unique page views for the product
Average time spent	Average time spent on the product's detail page

Table 3.1: Description of data values collected for each month/row

Metric	Mean	Std	Min	Max	Median
Sale Price (CZK)	1043.24	195.79	570.25	1251.73	1137.79
Stock price (CZK)	589.37	69.19	481.84	942.07	600.00
Stock (pcs)	20.84	19.12	0.00	166.00	20.00
Sold (pcs)	4.36	4.40	0.00	47.00	3.00
Total Earned (CZK)	4613.77	4790.42	0.00	51035.54	3400.12
Unique page views	66.37	62.67	3.00	1119.00	47.00
Average time spent (s)	59.24	29.84	10.14	366.84	53.83

Table 3.2: Training data metrics aggregated over all products and months

Metric	Mean	Std	Min	Max	Median
Sale Price (CZK)	957.37	288.93	528.93	1320.44	735.54
Stock price (CZK)	528.42	11.07	514.10	547.12	530.09
Stock (pcs)	19.93	17.37	0.00	46.00	12.00
Sold (pcs)	3.00	2.37	1.00	2.00	10.00
Total Earned (CZK)	3115.45	3166.31	528.93	13015.48	2204.38
Unique page views	38.22	27.43	4.00	125.00	30.00
Average time spent (s)	57.45	28.95	17.00	128.00	46.44

Table 3.3: Evaluation data metrics aggregated over all products and months

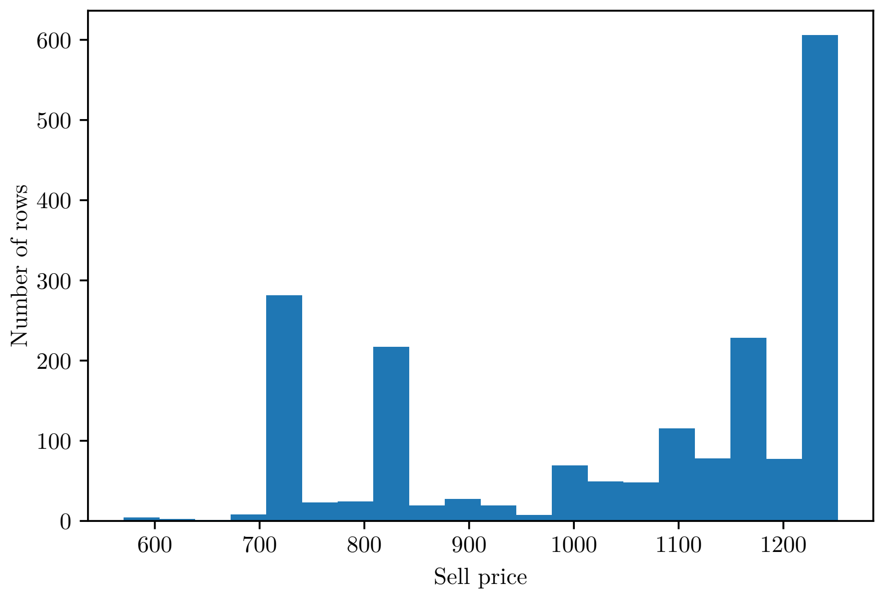


Figure 3.2: The histogram of sell prices from the training data

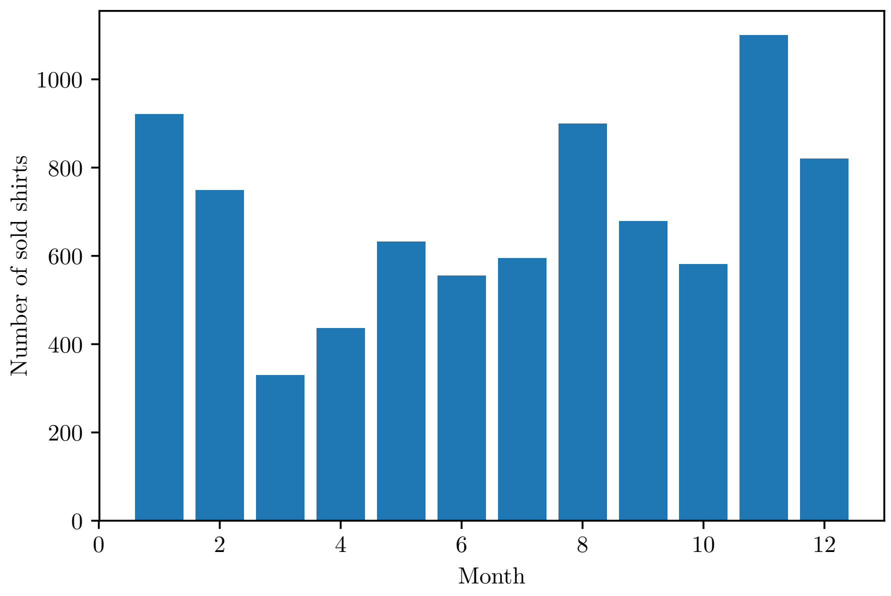


Figure 3.3: Number of sold shirts for every month from the training data

Chapter 4

Algorithms Overview

In this chapter, we look at the algorithms used in this work. We define the necessary prerequisites that enable us to build the algorithms. We begin with the definition of a Markov Decision Process (MDP) in Section 4.1. We will use this definition in Section 4.2 for the Reinforcement Learning (RL) problem introduction. We follow up on the topic of RL in Section 4.3 where we narrow our scope to RL using deep neural networks. Soft Actor-Critic (SAC) is the state-of-the-art off-policy algorithm for online RL. We provide a description of the algorithm in Section 4.4. The ability to learn from static datasets is very appealing, and we introduce the concept of Offline RL in Section 4.5. We need an algorithm for Offline RL to serve as the base for our dynamic pricing solution. Conservative Q-learning (CQL) is the main algorithm of choice for this work. We look at its details in Section 4.6. A sidestep to Imitation Learning (IL) happens in Section 4.7, we look at the Behavioral Cloning (BC) algorithm.

4.1 Markov Decision Process

In this section, we describe a formalism for sequential decision problems with uncertainty. The decision only depends on the current state and does not consider the past. Commonly used formalism for this kind of problem is an MDP.

An MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. Here, $\mathcal{S}$ is the *state space*, where the state $s_0 \in \mathcal{S}$ is the *initial state*, $\mathcal{A}$ is the *action space*, $\mathcal{P}$ is the *transition model*, $\mathcal{R}$ is the *reward function*, and $\gamma \in (0,1]$ is the *discount factor*. Note that $\mathcal{S}$ and $\mathcal{A}$ can be infinite.

We now introduce an agent that will observe a state of the environment $s_t \in \mathcal{S}$, where t is the time step. The agent performs an action $a_t \in \mathcal{A}$ based on the observed state s_t. The agent receives a reward $r_t = \mathcal{R}(s_t, a_t)$ for his action. This process continues until the environment reaches a *terminal state*.

A function that maps a state $s \in \mathcal{S}$ to an action $a \in \mathcal{A}$ is called a *policy*. The common symbol notation for a policy is $\pi : \mathcal{S} \to \mathcal{A}$. We get the action $a \in \mathcal{A}$ in a given state $s \in \mathcal{S}$ from our policy as $a = \pi(s)$. On a higher level, we can view a policy as an instance of the agent's behavior in the MDP.

Every useful policy needs to take the transition model into consideration. Generally, we consider the transition model $\mathcal{P}(s_{t+1}|s_t, a_t) \in [0,1]$ to be the probability of transitioning to state s_{t+1} when the action a_t is executed in the state s_t.

In Equation 4.1, we introduce a *utility function*. This function describes the cumulative expected reward in a state $s \in \mathcal{S}$ while following a policy π.

This utility function considers a so-called *finite horizon* MDP. A finite horizon MDP has a finite number of transitions T. An MDP with *infinite horizon* would have $T = \infty$. We usually consider $T = 12$ as twelve months, according to our domain description from Section 1.2.

The discount factor γ is usually tied to the infinite horizon MDPs. The reason behind this is that an agent could potentially exploit the rewards of the MDP forever. While this phenomenon cannot happen in a finite MDP, we include the discount factor to help us optimize the pricing process. It is better to sell the product earlier than later, because of the space it occupies in the warehouse. Thus, the agent prefers to sell the item sooner than later.

$$U^{\pi}(s) = \mathbb{E}\left[R_t | s_t = s\right], R_t = \sum_{k=0}^{T} \gamma^k \mathcal{R}(s_{t+k}, a_{t+k}) \tag{4.1}$$

Let us again look at the problem of maximization of the cumulative expected reward. We need an optimal policy for our agent. We denote it as π^*. The optimal policy maximizes the rewards and is recursively defined as follows (Equation 4.2).

$$\pi^*(s_t) = \underset{a_t \in \mathcal{A}}{\operatorname{argmax}} \sum_{s_{t+1} \in \mathcal{S}} \mathcal{P}(s_{t+1} \mid s_t, a_t) U^{\pi^*}(s_{t+1}) \tag{4.2}$$

If we have the knowledge of the transition model, we can find the optimal policy via Value or Policy iteration. We refer to [38] for an overview of Value and Policy iteration methods. We will discuss the methods of solving this problem without a transition model in Section 4.2 using RL.

4.2 Reinforcement Learning

In this section, we follow up on the definition of an MDP with RL. Generally, RL examines how an agent can learn from success and failure, reward, and punishment[38]. In other words, the agent is expected to learn from its own experience in an uncharted territory[41].

Let us consider an MDP, for which we do not know the exact transition model probabilities. An MDP with this property cannot be directly solved using the traditional methods mentioned in Section 4.1.

The main idea of RL is to learn a policy that maximizes the cumulative reward. We cannot use the direct transition model probabilities $\mathcal{P}(s_{t+1}|s_t, a_t)$, because these probabilities are unknown. This renders the Equation 4.2 useless. We need a different approach that does not require the transition model probabilities. Such approaches are called *model-free* methods. On the other hand, *model-based* methods learn the environment's transition model. We will not work with model-based methods in this work, we refer to [41] for more information.

While Equation 4.1 looks at the cumulative expected reward in each state $s \in \mathcal{S}$, we may want to look at the same property with respect to both state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$. This leads us to the definition of a *Q-function* in Equation 4.3.

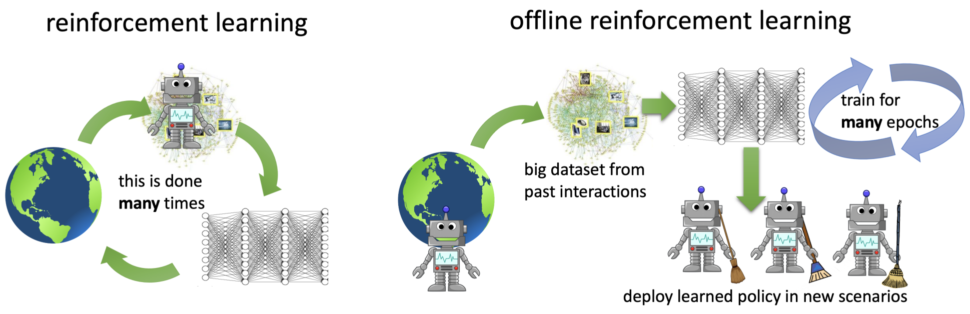

reinforcement learning
this is done many times
offline reinforcement learning
big dataset from past interactions
train for many epochs
deploy learned policy in new scenarios

Figure 4.1: Illustration of difference between online and offline RL methods [39]

$$Q(s_t, a_t) = \mathcal{R}(s_t, a_t) + \gamma \sum_{s_{t+1} \in \mathcal{S}} \mathcal{P}(s_{t+1} \mid s_t, a_t) \max_{a \in \mathcal{A}} Q(s_{t+1}, a) \tag{4.3}$$

Q-function has a direct relationship with the utility function. The relationship is shown in Equation 4.4.

$$U(s_t) = \max_{a_t \in \mathcal{A}} Q(s_t, a_t) \tag{4.4}$$

Equation 4.3 still includes the transition model probabilities. The solution to this is the *Q-learning* algorithm [47].

Q-learning uses a technique called *temporal difference* learning. This technique enables us to iteratively estimate the Q-function without the transition model. We obtain the estimation by "experience", the agent repeatedly experiences the environment and the transitions are used in the *Q-learning update*. We present the Q-learning update in Equation 4.5, the notation adopted from [38]. α is the learning rate.

The idea of environment exploration implies the ability of the agent to directly interact with the environment. This environment is referred to as the *simulator*. RL algorithms learning with a simulator available are called *online* algorithms. When there is no simulator available, we call these algorithms *offline*. We provide an illustration of the difference between online and offline RL in Figure 4.1. We take a look at Offline RL in Section 4.5.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(\mathcal{R}(s_t, a_t) + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{4.5}$$

A similar algorithm to Q-learning is called *SARSA*, which stands for State-Action-Reward-State-Action [38]. The difference versus Q-learning is in the choice of Q-value in the next state. While Q-learning takes the maximum over all possible Q-values in Q-learning update (Equation 4.5), SARSA chooses the actual Q-value observed while executing the current behavior policy. Because Q-learning does not take the actual policy into account, we call it an *off-policy* algorithm. SARSA is an *on-policy* algorithm, we need to observe the complete transition quintuple $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$, where $r_t = \mathcal{R}(s_t, a_t)$. The SARSA update (Equation 4.6) uses all elements of this quintuple.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(\mathcal{R}(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{4.6}$$

An important topic of RL is the trade-off between *exploration and exploitation* when performing subsequent “runs” in the environment. This problem arises due to the fact that the agent does not know the environment’s exact model and is learning the model’s estimates through interaction with the environment.

Exploration enables the agent to learn a better model of the environment, but the agent needs to exploit the states and actions that yield the highest rewards. A policy that would always exploit the current, possibly suboptimal, estimate and never explore new possibilities is a *greedy* policy.

One common approach to this trade-off is an *ϵ-greedy* policy. The agent performs a random action with a probability of ϵ. Otherwise, the agent greedily chooses the maximizing action according to the Q-values.

Q-learning in its basic form only works for discrete state spaces and actions. This limitation, paired with the so-called *curse of dimensionality*, renders it useless for our problem because we are dealing with continuous state and action spaces. We can handle continuous spaces by using function approximators on top of Q-learning. Current state-of-the-art function approximators are based on *deep neural networks*. We look into DRL in the next section 4.3.

4.3 Deep Reinforcement Learning

In this section, we introduce Deep Learning and apply its methods to RL. Specifically, we use deep neural networks to represent the Q-function. We use this to introduce the Deep Q-network (DQN) algorithm.

4.3.1 A Quick Introduction to Deep Learning

Deep Learning uses deep neural networks as function approximators to solve challenging tasks in machine learning. We usually build a deep neural network from an input layer, an arbitrary number of hidden layers, and an output layer. The number of hidden layers is considered the *depth* of the network. The term "Deep Learning" arose from this terminology [13].

Neurons are the building blocks of each layer. The number of neurons per layer is the *layer size*. The most common *linear layers* consist of neurons that represent an affine transformation. We show the affine transformation in Equation 4.7, where $x \in \mathbb{R}^n$ is the neuron input, $w \in \mathbb{R}^n, b \in \mathbb{R}$ are the parameters, and $y \in \mathbb{R}$ is the neuron output.

$$y(x) = w^T x + b \tag{4.7}$$

Naturally, a deep neural network that only consists of linear layers is linear. We would like to approximate nonlinear functions. To introduce nonlinearity to the network, we add *activation functions*. We use the activation function after each linear layer. The most common activation functions are:

- Rectified Linear Unit (ReLU) $y(x) = \max(0, x)$
- Logistic Sigmoid $y(x) = \frac{1}{1+e^{-x}}$
- Hyperbolic Tangent $y(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$

Equation 4.7 includes parameters w and b. However, so far, we have not described a way to learn these parameters. It is important to note that deep neural networks have many parameters, and it is common to denote them all combined as θ.

Deep neural networks learn their parameters from *training data*. This data consists of the input data and its ground truth values. We pass the data through our network and get some output. We call this step the *forward pass*. The next step is to compute the network's *loss*. The loss describes the error between ground-truth values and the network's predictions. The last step tries to improve the network's parameters, we call this step *backward pass*. The general algorithm is called *Backpropagation*.

Backward pass computes the gradients via the chain rule, beginning with the loss function and then starting at the last layer. Hence, the name backward pass. We use the calculated gradients to update the network's parameters. This concludes a single training step.

A method for gradient computation is a pivotal part of Deep Learning. Currently, methods based on Stochastic Gradient Descent (SGD) with momentum are prominent. The state-of-the-art method used nowadays is ADAM[21] or its variants.

While this Deep Learning overview may be quite brief, anything more detailed would fall outside the scope of this work. We refer the reader to [13] for more information about Deep Learning.

4.3.2 Deep Q-network

The introduction of DQN[31] is the beginning of the DRL era. DQN combines RL and Deep Learning. Here, we describe the improved DQN version from [32]. A deep neural network represents the Q-function. RL usually diverges when paired with a nonlinear function approximator to represent the Q-function. The mitigation of this problem is done by two key ideas.

The first idea is the usage of *experience replay*[28]. The agent stores his experience in the form of a *transition* (s_t, a_t, r_t, s_{t+1}) into a Replay Buffer. During training, a batch of samples is drawn from the Replay Buffer in a uniform fashion. This technique helps to break the correlation between observation sequences and improves data efficiency by reusing previous transitions.

The second idea introduces a second deep neural network to help stabilize the learning. We refer to this network as the *target network* $\hat{Q}$, the notation adopted from [32]. The target network $\hat{Q}$ begins with the same parameters as the main network Q. The difference is the update frequency. While the main network Q gets updated every training step, the target network $\hat{Q}$ only gets updated once every C training steps. The actual update just copies the parameters from Q to $\hat{Q}$. We use $\hat{Q}$ in the Q-learning update adapted for DQN (Equation 4.8).

Comparing Equation 4.8 and Equation 4.5, we see that the only relevant change is the usage of $\hat{Q}$ in the learning part of the equation. One other subtle difference is the introduction of θ and θ'. These symbols represent the learnable parameters for each Q-function network.

$$Q(s_t, a_t; \theta) \leftarrow Q(s_t, a_t; \theta) + \alpha(\mathcal{R}(s_t, a_t) + \gamma \max_{a \in \mathcal{A}} \hat{Q}(s_{t+1}, a; \theta') - Q(s_t, a_t; \theta)) \tag{4.8}$$

The "learning" part of the Q-learning update is often called the *Temporal Difference error*. Temporal Difference error is shown in Equation 4.9.

$$\mathcal{R}(s_t, a_t) + \gamma \max_{a \in \mathcal{A}} \hat{Q}(s_{t+1}, a; \theta') - Q(s_t, a_t; \theta) \tag{4.9}$$

We show the actual loss function of the neural network in Equation 4.10 (notice the usage of the Temporal Difference error from Equation 4.9). We then perform the standard backpropagation procedure (See Section 4.3.1 for more information). For more specific details, please refer to [32].

$$L(\theta) = \mathbb{E}_{s_t, a_t, s_{t+1} \sim \mathcal{D}} \left[(\mathcal{R}(s_t, a_t) + \gamma \max_{a \in \mathcal{A}} \hat{Q}(s_{t+1}, a; \theta') - Q(s_t, a_t; \theta))^2 \right] \tag{4.10}$$

While DQN made a breakthrough in RL and is a fundamental algorithm for DRL, nowadays, there are algorithms that have surpassed its performance. Soft Actor-Critic (SAC) is the state-of-the-art algorithm for DRL. We describe SAC in the next section 4.4.

4.4 Soft Actor-Critic

In this section, we introduce the family of policy gradient methods, restrict their scope to actor-critic methods, and then we describe the Soft Actor-Critic (SAC) algorithm.

Policy Gradient Methods

Up until now, all methods described for solving the RL problem used the iterative estimate of the Q-function (Equation 4.5) to train a policy. The *policy gradient* methods take a different approach. They learn a *parametrized policy* without using a Q-function for action selection.

We need to define a parametrized policy. Up until now, we only considered *deterministic* policies. A deterministic policy is a direct mapping from a state $s \in \mathcal{S}$ to an action $a \in \mathcal{A}$. A *stochastic* policy is defined in Equation 4.11. Each state-action pair has a probability to be chosen by the policy.

$$\pi(a|s) : \mathcal{A} \times \mathcal{S} \rightarrow [0, 1], a \in \mathcal{A}, s \in \mathcal{S} \tag{4.11}$$

Coming back to the parametrized policy, we define a parametrized policy as a stochastic policy, based on a parameter $\theta \in \mathbb{R}^d$, in Equation 4.12, the notation inspired by [41].

$$\pi(a|s, \theta) : \mathcal{A} \times \mathcal{S} \times \mathbb{R}^d \rightarrow [0, 1], a \in \mathcal{A}, s \in \mathcal{S}, \theta \in \mathbb{R}^d \tag{4.12}$$

The name “Policy Gradient Methods” implies the usage of gradient. Here, we will follow the notation from [41]. To specify this, we introduce a scalar measure $J(\theta)$ with respect to the policy parameter θ. We want to maximize the performance, so we perform gradient **ascent** in Equation 4.13. α is the step size. $\widehat{\nabla J(\theta_t)}$ is a stochastic estimate of $\nabla J(\theta)$ with respect to θ_t.

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)} \tag{4.13}$$

Methods using this general idea of gradient ascent are called policy gradient methods. We need to obtain the estimated gradient of the scalar measure $\nabla J(\theta)$ with respect to the policy parameter θ. Fortunately, an answer for this problem exists. The answer is the *policy gradient theorem*, shown in Equation 4.14. μ_π is the state distribution with respect to the policy π. The state distribution essentially represents the fraction of time spent in every state $s \in \mathcal{S}$ while following the policy π. $q_\pi(s, a)$ is the value of taking an action $a \in \mathcal{A}$ in a state $s \in \mathcal{S}$ under policy π.

$$\nabla J(\theta) \approx \sum_{s \in \mathcal{S}} \mu_\pi(s) \sum_{a \in \mathcal{A}} q_\pi(s, a) \nabla \pi(a|s, \theta) \tag{4.14}$$

Actor-Critic Methods

The *actor-critic* methods are a subset of policy gradient methods. They draw ideas from temporal difference methods, e.g. Q-learning (Equation 4.5). We could say that actor-critic methods combine policy gradient methods and temporal difference methods. One example of using actor-critic methods combined with neural networks is the A3C[30].

We will build upon the basic definition of the policy gradient theorem (Equation 4.14). We define a generalized policy gradient theorem with a baseline $b(s) : \mathcal{S} \rightarrow \mathbb{R}$ in Equation 4.15. For the proof of correctness, see [41]. We use the baseline to determine whether the returns are better or worse than average.

$$\nabla J(\theta) \approx \sum_{s \in \mathcal{S}} \mu_\pi(s) \sum_{a \in \mathcal{A}} (q_\pi(s, a) - b(s)) \nabla \pi(a|s, \theta) \tag{4.15}$$

A good choice for the baseline is the utility function (Equation 4.1). We cannot get the exact utility, so we have to settle for an estimate. We obtain the utility estimate $\hat{U}_\pi(s)$ via supervised learning. This estimate is the *critic* part of actor-critic methods. For completeness, the *actor* is the actual policy. Specifically, we use a neural network that approximates the utility function of the most recent policy π. The utility estimate is concurrently updated with the policy while training.

The network is learned by standard SGD methods using the mean squared error of the temporal difference error. The actual objective minimizes the difference between the reward of a transition sampled from the current policy π, and the estimated utility function. We define the *temporal difference error* in Equation 4.16. We assume that we sampled a transition tuple (s_t, a_t, r_t, s_{t+1}). Let us define $r_t = \mathcal{R}(s_t, a_t)$ to simplify the notation.

$$\delta = (r_t + \gamma \hat{U}_\pi(s_{t+1})) - \hat{U}_\pi(s_t) \tag{4.16}$$

Maximum Entropy Reinforcement Learning

Maximum entropy RL augments the standard RL objective with an entropy term. In this section, we follow the notation from [14], the article proposing the maximum entropy RL. We define the standard optimal policy in Equation 4.17. π is the policy. μ_π is the state distribution of trajectories.

$$\pi^*_{std} = \arg\max_\pi \sum_t \mathbb{E}_{(s_t,a_t)\sim\mu_\pi}[\mathcal{R}(s_t, a_t)] \tag{4.17}$$

Equation 4.17 augmented with the entropy term is shown in Equation 4.18, where α determines the relative importance of the entropy term, we refer to α as the *temperature*, $\mathcal{H}(\pi(\cdot|s_t))$ is the entropy of the action distribution in state s_t for the policy π.

$$\pi^*_{MaxEnt} = \arg\max_\pi \sum_t \mathbb{E}_{(s_t,a_t)\sim\mu_\pi}[\mathcal{R}(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))] \tag{4.18}$$

We define the *soft Q-function* in Equation 4.19. We use $r_t = \mathcal{R}(s_t, a_t)$ for simpler notation.

$$Q^*_{soft}(s_t, a_t) = r_t + \mathbb{E}_{(s_{t+1},\dots)\sim\mu_\pi}\left[\sum_{k=1}^{\infty} \gamma^k (r_{t+k} + \alpha\mathcal{H}(\pi^*_{MaxEnt}(\cdot|s_{t+k})))\right] \tag{4.19}$$

For more information and proofs, see [14]. We use a soft Q-function approximated by a neural network in the SAC algorithm.

Clipped Double Q-learning

The idea of *clipped double Q-learning* comes from the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm[11]. This technique is another tool that improves learning stability. Specifically, clipped double Q-learning helps to prevent *overestimation bias*. This overestimation bias happens due to the fact that the critic is an estimate of the true function.

We need to clarify one thing. In Equation 4.15, we have the term $q_\pi(s,a) - b(s)$. This term is the standard policy gradient theorem with a baseline, which is very general. Equation 4.16 replaces this with a term called *one-step actor-critic*, sometimes named *TD actor-critic*. From now on, we will only consider so-called *Q actor-critic*. Essentially, this actor-critic uses a Q-function estimate as a critic. Proof can be found in [1].

Let us now consider a critic $Q_\theta(s,a)$. This critic is an estimate of the Q-function (Equation 4.3). We approximate the Q-function using a neural network. θ denotes the network's parameters. We use the stabilization trick from DQN, using a second network called target network. We denote its parameters by θ'.

The authors of [11] show that overestimation bias is an issue. The method of mitigation is simple. Instead of using a single network, we introduce a second one. The same is done with the target network. We then denote the parameters as $\theta_1, \theta_2, \theta'_1, \theta'_2$.

These networks are learned independently using standard SGD methods. The interesting part happens in the training of these networks. Their loss is calculated as the mean squared error. When we calculate this loss, we use the minimum of those two networks. The formula of the general idea is $\min_{i=1,2} Q_{\theta'_i}(s_{t+1}, a_{t+1})$. This reduces the overestimation bias. We refer to [11] for technical details. We use clipped double Q-learning in the SAC algorithm.

Soft Actor-Critic

Up until now, we have been building up the theory and ideas to build the state-of-the-art RL algorithm Soft Actor-Critic (SAC). Now, we define the SAC algorithm, connect it with previous ideas, and describe its attributes.

A side note, the authors of SAC introduced the first version in [15]. This algorithm used an explicit utility function estimate and worked with a fixed temperature parameter. The authors introduced an updated version [16]. This version does not work with a utility function estimate and has automatic temperature parameter tuning.

SAC is an online, model-free, off-policy RL algorithm that works with continuous state and action spaces. The algorithm's main new idea is the maximum entropy RL 4.4. SAC uses two core ideas from DQN, namely, experience replay and target networks, see Section 4.3.2 for more information. The last core idea is the usage of clipped double Q-learning 4.4.

We show the algorithm pseudocode in Algorithm 1. This pseudocode is mostly copied from [16] with some minor variations to keep the notation consistent. We follow up with a high level description of the algorithm.

First, we need to describe all parameters. ϕ are the policy weights. A neural network represents the policy. Four more neural networks represent the Q-functions. Weights θ_1, θ_2 are for the main networks and the weights θ'_1, θ'_2 are for the target networks. α is the temperature parameter. The next mentioned parameters are constant. We call them *hyperparameters*. All λ parameters are the step sizes for the SGD. τ controls the speed of target network learning.

We now look at the actual algorithm steps. The first step is the initialization. We have some initial parameters θ_1, θ_2, ϕ. We copy the Q-function weights into the target networks. We start with an empty Replay Buffer $\mathcal{D}$.

We now perform training iterations. It is common to use a fixed number of iterations while training, commonly referred to as *epochs*.

Every iteration, we perform *policy rollouts*. Policy rollouts take the current policy π and sample actions that are forwarded to the environment. The complete transition tuples are then stored in the Replay Buffer $\mathcal{D}$.

Algorithm 1: Soft Actor-Critic

```
Input: $\theta_1, \theta_2, \phi$                         ▷ Initial parameters
$\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2$       ▷ Initialize target network weights
$\mathcal{D} \leftarrow \emptyset$                        ▷ Initialize an empty replay buffer
for each iteration do
    for each environment step do
        ▷ Sample action from the policy
        $a_t \sim \pi_\phi(a_t|s_t)$
        ▷ Sample transition from the environment
        $s_{t+1} \sim \mathcal{P}(s_{t+1}|s_t, a_t)$
        ▷ Store the transition in the replay buffer
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, \mathcal{R}(s_t, a_t), s_{t+1})\}$
    end for
    for each gradient step do
        ▷ Update the soft Q-function weights
        $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
        ▷ Update policy weights
        $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
        ▷ Adjust temperature
        $\alpha \leftarrow \alpha - \lambda_\alpha \hat{\nabla}_\alpha J(\alpha)$
        ▷ Update target network weights
        $\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$ for $i \in \{1, 2\}$
    end for
end for
Output: $\theta_1, \theta_2, \phi$                        ▷ Optimized parameters
```

Naturally, we have to update the learnable parameters to see improvement. For the sake of simplicity, we will not include exact gradient formulas for the updates. For information on the formulas, see [16]. We update $\theta_1, \theta_2, \phi, \alpha$ via SGD. Batch of data for SGD is sampled from the Replay Buffer. The weights for the target networks, θ'_1, θ'_2, are updated using *polyak averaging* with respect to the hyperparameter τ.

This concludes the description of SAC. Our problem is not solved by SAC only, but we use it as the base for the CQL algorithm, described in Section 4.6.

4.5 Offline Reinforcement Learning

The idea of learning strong policies via a generalized algorithm that does not need fine-tuning for specific tasks is very appealing. For this reason, RL is a hot research topic. However, a major issue prevents the adoption of RL in real-world environments. Most real-world domains do not enable online collection of data. The data collection is either expensive or risky.

This motivated a new branch of RL, *Offline Reinforcement Learning*, sometimes referred to as batch RL. Offline RL tries to learn an optimal policy only from prior data without any online interaction with the environment. The motivation behind this is the success of supervised learning in domains like computer vision, where neural networks learn from large previously collected datasets and do not require any other data. Offline RL builds on top of DRL algorithms, so we can consider Offline RL as a subfield of DRL.

It is important to realize that learning from offline datasets creates a lot of challenges. The primary challenge is the *distributional shift*. Offline dataset does not guarantee that the distribution of its data corresponds to the real distribution. This produces an overestimation error when the evaluated state varies from the offline dataset. The agent would choose a suboptimal action that has overestimated Q-function. This renders conventional online RL methods useless in the setting of a fixed dataset. The attempt to fix this issue tries to restrict the policy so that the policy only chooses actions that are in the dataset or close to these. Restrictions on the policy cripple the generalization capabilities. Algorithms like BCQ, BEAR choose such a restrictive approach.

The state-of-the-art Offline RL algorithm Conservative Q-learning (CQL)[24] takes a different approach. CQL learns a conservative Q-function that lower bounds its true value. In practice, CQL only adds a slight change to the learning objective of online RL algorithms. In this work, we use CQL on top of SAC. We describe CQL in Section 4.6.

4.6 Conservative Q-Learning

In this section, we look at the state-of-the-art Offline RL algorithm CQL[24]. It is important to note that CQL is not an exact algorithm, a better description would be an algorithmic framework. The primary issue that happens while learning from a fixed dataset is the distributional shift. A Q-function trained on this dataset will overestimate unseen actions. The primary idea behind CQL is to learn a conservative Q-function that lower bounds the true Q-function. We provide an illustration of the conservative Q-function idea in Figure 4.2.

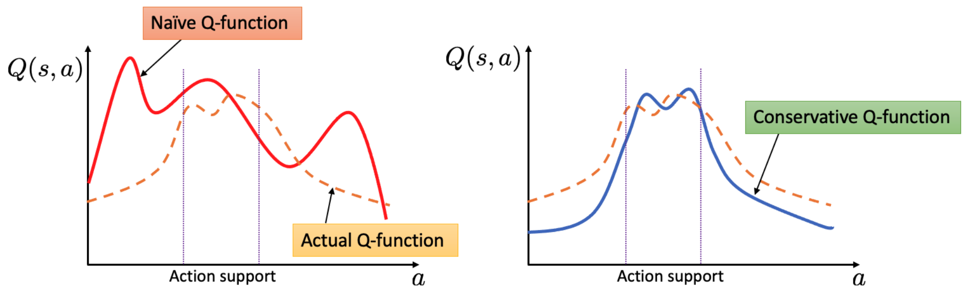


Figure 4.2: Illustration of the conservative Q-function [39]

We build CQL on top of SAC in Algorithm 2. For an overview of SAC, see Section 4.4.

Algorithm 2: Conservative Q-learning using SAC

```
Input: $\theta_1, \theta_2, \phi, \mathcal{D}$        ▷ Initial parameters and replay buffer
  $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2$        ▷ Initialize target network weights
  for each iteration do
    for each gradient step do
      ▷ Update the soft Q-function weights
      $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} CQL(\theta_i)$ for $i \in \{1, 2\}$
      ▷ Update policy weights
      $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
      ▷ Adjust temperature
      $\alpha \leftarrow \alpha - \lambda_\alpha \hat{\nabla}_\alpha J(\alpha)$
      ▷ Update target network weights
      $\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$ for $i \in \{1, 2\}$
    end for
  end for
Output: $\theta_1, \theta_2, \phi$        ▷ Optimized parameters
```

Next, we look at the differences between vanilla SAC and CQL. First, the Replay Buffer $\mathcal{D}$ is an input, instead of being an empty set. We prefill the Replay Buffer $\mathcal{D}$ with the previously collected transitions from our offline dataset. The transition collection is also missing. We do not collect new trajectories in Offline RL. The optimization step remains unchanged for the most part, but there is a change in the Q-function update objective. The change is highlighted in red in Algorithm 2.

We use the CQL objective here, which is given in Equation 4.20. α is the temperature parameter, $\hat{\pi}_\beta$ is an approximation of the original policy from the offline dataset, and $\mathcal{D}$ is the Replay Buffer. The minimizing term is the CQL regularizer, which prevents the overestimation of unseen actions, the second term is the standard mean squared difference of the temporal difference error (see Equation 4.10).

$$\begin{aligned} CQL(\theta) = \min_Q \alpha \mathbb{E}_{s_t \sim \mathcal{D}} \left[\log \sum_{a_t} \exp(Q(s_t, a_t; \theta)) - \mathbb{E}_{a_t \sim \hat{\pi}_\beta(a_t|s_t)} \left[Q(s_t, a_t; \theta) \right] \right] \\ + \mathbb{E}_{s_t, a_t, s_{t+1} \sim \mathcal{D}} \left[\left(\mathcal{R}(s_t, a_t) + \gamma \max_{a \in \mathcal{A}} \hat{Q}(s_{t+1}, a; \theta') - Q(s_t, a_t; \theta) \right)^2 \right] \end{aligned} \tag{4.20}$$

For more information on the CQL objective and theoretical background with proofs, see [24].

4.7 Imitation Learning

In this section, we provide an introduction to Imitation Learning (IL). We then introduce the Behavioral Cloning (BC) algorithm.

Introduction to Imitation Learning

We use Imitation Learning (IL) to learn a desired behavior by imitating the expert's behavior [34]. IL is closely related to RL, specifically because they both use the MDP formalism. However, there is a fundamental difference that separates IL and RL.

IL learns a policy by mimicking the provided expert demonstrations, and therefore it is not able to generalize in any sense. On the other hand, RL is able to generalize. This creates a difference in the reward interpretation. While RL uses the reward as the driving metric for training. The main motive of IL is to mimic the provided behavior given by the expert.

4.7.1 Behavioral Cloning

The simplest variant of IL is Behavioral Cloning (BC). BC learns a direct mapping from a state $s \in \mathcal{S}$ to an action $a \in \mathcal{A}$. Because of this fact, the BC task is a simple supervised learning regression. We implement this regression task as a deep neural network (see 4.3.1 for a quick intro to Deep Learning). We denote the network's parameters by θ. Furthermore, we describe the neural network as a deterministic policy in Equation 4.21.

$$\pi(s;\theta) : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathcal{A}, s \in \mathcal{S}, \theta \in \mathbb{R}^d \tag{4.21}$$

We perform the optimization of the network's parameters θ via SGD. The loss function is simply the mean squared error of the action given by the neural network and the expert's action. We denote the expert's behavior by a policy $\pi_E(s) : \mathcal{S} \rightarrow \mathcal{A}, s \in \mathcal{S}$. We perform the optimization on batches of states, we denote a single batch by $\mathcal{B}$. Finally, we give the loss function of the neural network in Equation 4.22.

$$L(\theta) = \mathbb{E}_{s \sim \mathcal{B}} \left[(\pi(s;\theta) - \pi_E(s))^2 \right] \tag{4.22}$$

In practice, we use the RL toolkit Garage[12], which has an implementation of BC based on [18].

We use CQL and BC for dynamic pricing in this work. We describe the underlying MDP in Chapter 5. Lastly, we evaluate our results via domain expert evaluation and simulation in Chapter 6.

Chapter 5

Model

In this chapter, we provide a model of the underlying MDP used for dynamic pricing (Section 5.1), and we describe a custom simulator used in Chapter 6 for evaluation (Section 5.2).

5.1 Dynamic Pricing Model

We described the general domain of this work in Chapter 1, we introduced the formalism of MDPs in Section 4.1. Furthermore, we combine the MDP formalism and the domain description to create a concrete MDP model. We draw inspiration from [29], but the data available is the deciding factor in most cases. We follow up with the description of three pivotal parts of the model, namely, the state space $\mathcal{S}$, the action space $\mathcal{A}$, and the reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$.

State Space

We use a continuous state space $\mathcal{S}$, we consider a single state $s \in \mathcal{S}$ to be a vector of carefully selected features. All elements of the state space have a dimension of eight, and we consider all elements of the vector to be real numbers. Thus, we say that $s \in \mathbb{R}^8$.

By doing this, we introduce inaccuracies, because some features are discrete numbers, e.g., month in the year, count of products in stock. Nevertheless, an implementation of a state space that has some discrete and some continuous elements is not a common approach. Therefore, we proceed with the usage of a continuous state space, but it should be noted that some features only have a discrete set of possible values.

It is important to realize that all features describe the dynamics of a single month in the e-commerce store. We proceed to describe every used feature and give some motivation behind the particular choice. The feature vector has eight dimensions, we start with the first dimension, the month. This dimension represents the calendar month of the year that the state happened in. It makes sense to have the month as one of the features, because the customer behavior changes during the year.

The second dimension is the number of sold shirts. We use this number to determine the exact number of sold shirts, and we subtract this number from the stock count. The total revenue is the third dimension of the state vector, it represents the sheer amount of money that was gained from the given month of sales. We use the CZK currency for all money-related features.

The actual selling price of the shirt is the fourth dimension, this represents the price that is visible to customers when they browse the store's website. The fifth dimension represents the stock price. This is the price that the e-commerce store bought the shirt from its supplier. We use the stock price for profit calculation in the reward function. Naturally, we should keep track of the stock in the warehouse, and this number is stored in the sixth dimension of the feature vector.

The last two dimensions of the feature vector represent the data from Google Analytics (GA). We store the unique page views in the seventh dimension. This metric describes the unique number of visits to the product's detail page. The unique number of visits is also used in the reward function. The last dimension is the average time spent on the product's detail page. We decided not to use this feature in the reward function.

We do not include any exact metrics here, for data metrics, see Table 3.2.

Action Space

The action space we use is continuous and has one dimension. The action $a \in \mathcal{A}$ is the price choice for the next month. We have to choose a lower and upper bound for the price.

A natural choice for the bounds is the maximum and minimum of the prices from the training data. We decide to choose the bounds as 500 and 1500 for the lower and upper bounds, respectively. The chosen bounds for prices create a larger action space than the action space induced by the maximum and minimum of the training data. This allows for better flexibility of the pricing framework.

Reward Function

The choice of the reward function is important. A general first choice may be the total revenue or profit from the sales. This is a short-sighted choice in the e-commerce domain, because the customer demand could drastically fluctuate.

A reward that takes demand into account is preferable. Fortunately, we can get a pretty good demand estimate from the GA data. Specifically, we use the unique page view metric as the demand estimate.

When we take the revenue or profit and divide it by the unique page views, we obtain the reward function. We consider this to be a new artificial metric called *conversion rate*. We tried to use the revenue conversion rate, but we did not achieve reasonable results. Specifically, the agent started to choose low prices with the motivation to sell as much as possible, and the actual profit was low. For the rest of this work, we use the profit conversion rate as the reward function, as shown in Equation 5.1.

$$conversionRate = \frac{profit}{uniquePageViews} \tag{5.1}$$

5.2 Simulator

We choose to base our simulation on [19] because the authors proposed a straightforward and interpretable simulation. Specifically, we use a Poisson process to model the general customer demand, and we use a uniform distribution spanning over the whole action space to model the acceptable price.

This simulation is a finite horizon MDP. We consider the horizon $T = 12$, which corresponds to one year (one step corresponds to one month). The simulation can terminate in fewer steps if the stock gets sold out.

We implement the simulation in OpenAI's Gym[2], written in Python, for exact implementation details and metrics, see Section 6.2.

Chapter 6

Experiments

In this chapter, we evaluate the proposed methods, compare them with baselines, discuss the observed phenomenons, and provide strengths and weaknesses. We examine the domain expert evaluation in Section 6.1. Next, we dive into results of the simulation in Section 6.2.

6.1 Domain Expert Evaluation

In this section, we describe the methodology and results of an evaluation performed by a domain expert. Specifically, the evaluator is the chief financial officer of the business that provides us with real-world data. Due to the exclusivity of the products to a single seller, the number of possible domain experts is small. Thus, the domain expert that performed the evaluation is the best possible evaluator for the data we have. Next, we describe the evaluation methodology.

6.1.1 Evaluation Methodology

We propose a simple evaluation methodology that aims to reduce possible bias, the methodology consists of three main parts:

- evaluation instructions,
- method of evaluation results collection,
- sample selection and randomization.

Evaluation Instructions

To help reduce the possibility of bias in the evaluation results, we gave a strict set of rules to the evaluator to ensure maximum objectivity:

- the evaluation must be performed in one take, without any distractions,
- the domain expert must evaluate every sample independently, with equal effort, and
- there is no option to reevaluate any of the samples.

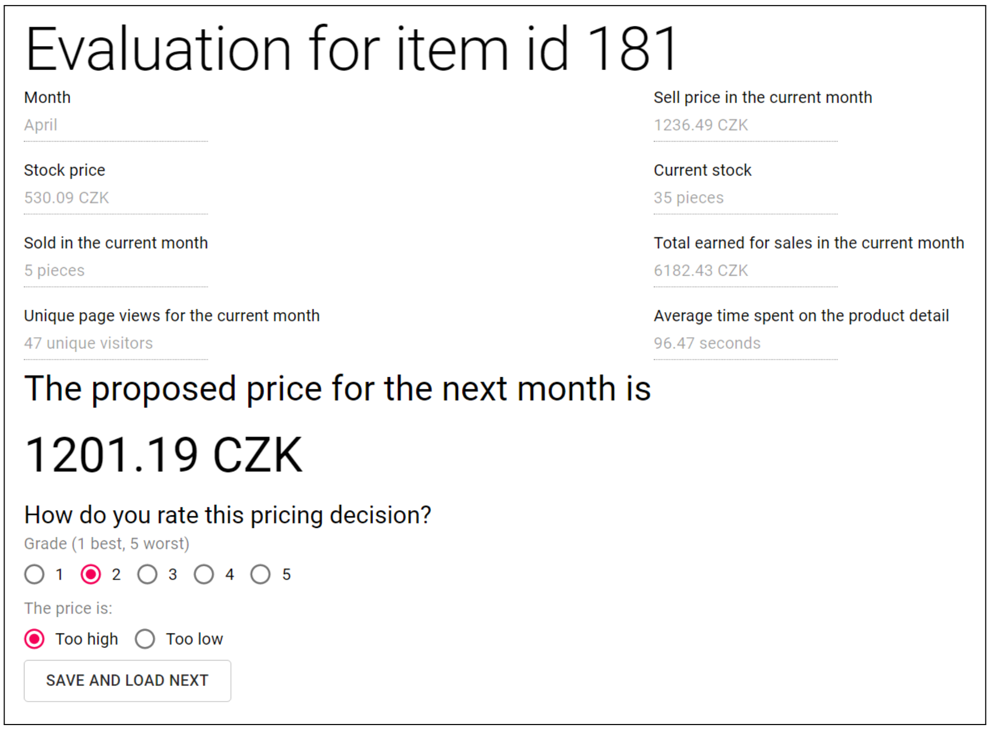
Evaluation Collection

We ensure a smooth evaluation results collection by introducing a simple web interface, we used React (JavaScript) for the frontend, FastAPI (Python) for the backend, and PostgreSQL as the database. The main objective of the interface was to provide a simple, yet elegant method for domain expert evaluation. The website consists of two pages, the login page and the evaluation page.

The login page is rather simple, the user only uses it to log in based on credentials. The evaluation page is the main part of the web interface, the user uses it to evaluate the samples from the dataset. We show a sample screenshot from the evaluation page in Figure 6.1.

The website shows the user all features of the state space for the current month (see Section 5.1 for more information on the features). The website also shows the proposed price for the next month coming from one of our algorithms. The user decides whether the choice is good or not but is not told what method was used to generate the price.

We use a discrete set of "grades" for the ranking, lower means better. The actual set of grades is $\{1, 2, 3, 4, 5\}$. For grades lower than one, we also use a helper metric called "pricing direction", this metric has two choices: too high or too low. This means that the proposed choice sets the price too high or too low.



Evaluation for item id 181

Month: April	Sell price in the current month: 1236.49 CZK
Stock price: 530.09 CZK	Current stock: 35 pieces
Sold in the current month: 5 pieces	Total earned for sales in the current month: 6182.43 CZK
Unique page views for the current month: 47 unique visitors	Average time spent on the product detail: 96.47 seconds

The proposed price for the next month is

1201.19 CZK

How do you rate this pricing decision?

Grade (1 best, 5 worst)

1 [✓ 2] 3 4 5

The price is:

[✓ Too high] Too low

Figure 6.1: The evaluation page of the web interface used for expert evaluation

Sample Selection and Randomization

We randomize the data for evaluation with respect to the used policies. Specifically, we used four different policies to evaluate the data and randomized the order of the resulting dataset. We described the evaluation dataset in Section 3.2, this dataset only consists of 45 samples. While this may seem like a small dataset, we have to take the time required for evaluation into consideration. Furthermore, with the usage of four different policies, the total number of evaluated samples is 180. The evaluator was not aware of the fact that different policies were used.

6.1.2 Expert Evaluated Pricing Policies

We now describe the four pricing policies used in the evaluation process. The first policy we used is the random policy. This policy just samples a random action from the uniform distribution given by the action space. We use this policy as a baseline, because it feels reasonable to expect nonrandom policies to outperform a random one.

The second and third policies use the BC algorithm (see Section 4.7.1). The difference between the two BC policies is the training set. We trained the first policy (denoted "BC (seasonal)") using a reduced dataset including only seasonal shirts. This dataset consists of 590 samples, which is about three times less than the full training dataset described in Section 3.2. We trained the second policy (denoted "BC") on the whole dataset. We managed to train good policies on both of these datasets, however, due to the nature of the BC algorithm, we cannot expect these policies to have any generalization properties.

The last policy uses the CQL algorithm (see Section 4.6) trained on the full training dataset described in Section 3.2. We did not manage to stabilize learning on the reduced dataset including only seasonal shirts. CQL needs a large enough and diverse dataset to learn a reasonable policy, we explore this problem in more detail in Section 6.2. We expect that the CQL algorithm trains a policy that is able to generalize.

6.1.3 Results of Expert Evaluation

The actual results of the domain expert evaluation are shown in Figure 6.2. The grades show that all trained algorithms outperformed the random policy. While the average results show the general performance of the evaluated policies, we do not see much detail.

The grades were not the only collected evaluation metric, we also collected "pricing directions". The results show that all algorithms usually overestimated reasonable prices as set by the domain expert. We show a histogram of pricing directions in Figure 6.3. Looking back at Figure 3.2, we can see that our dataset is biased towards higher prices, this provides some explanation on why are we observing such overestimation phenomenon. We back this fact up with Figure 6.4, which shows the distribution of proposed prices for each policy.

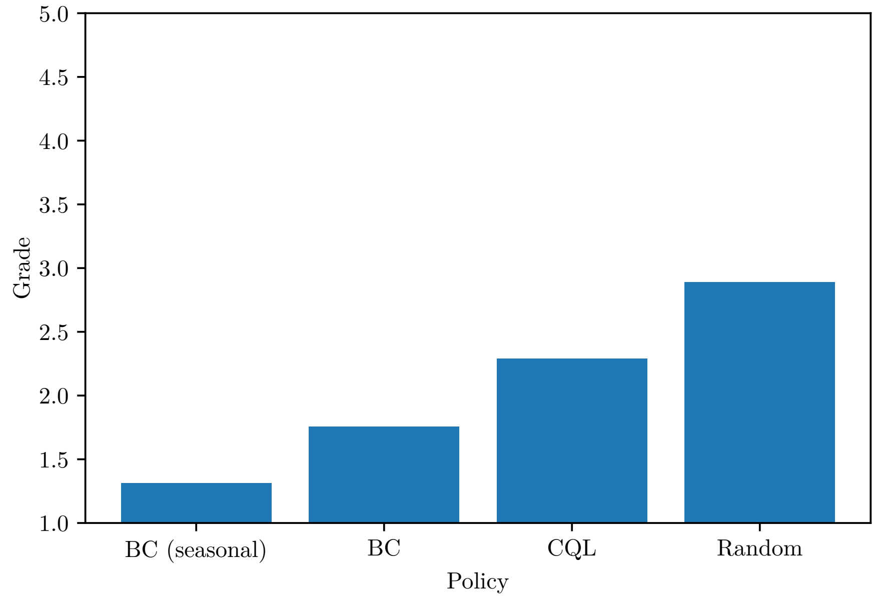


Figure 6.2: Average grade for used policies in the expert evaluation (lower is better)

Furthermore, we can see that BC trained on seasonal data only shows a much more conservative policy in terms of maximum proposed prices and in terms of the median. We note that this is expected, after all, BC simply learns to mimic an expert's behavior, in this case, BC mimics the historical seasonal data pricing. A future direction with a more diverse dataset could enable us to train even better policies.

Nevertheless, the results of the domain expert evaluation still look promising, we show a detailed histogram of all grades for every policy in Figure 6.5. We can see that nonrandom policies achieve great results with respect to the small size of the training dataset. While the CQL algorithm does not perform as well as the BC variants, this is expected due to the fact that CQL requires a large dataset to train well, and the results it achieves considering the size of our dataset are still great.

6.2 Simulation Results

In this section, we evaluate the results we obtained from experiments performed on our custom simulator. We provide a high-level introduction to the

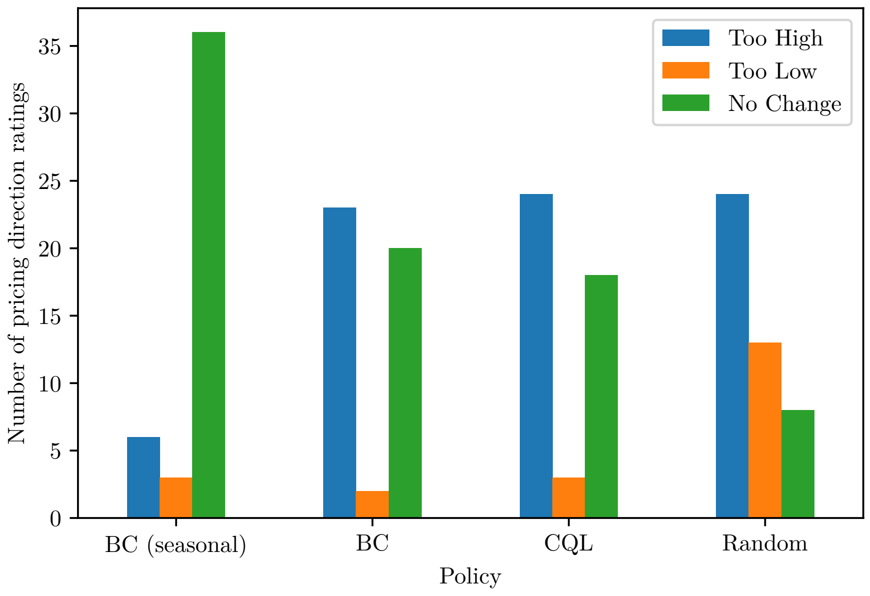


Figure 6.3: Histogram of pricing directions from the expert evaluation

simulator in Section 5.2. Next, we follow up with the exact implementation details.

Simulator Implementation Details

We implement the simulator in OpenAI's Gym[2]. Specifically, we create a Python package named *gym-pricing* that includes the implementation of the pricing environment. The interface for the package is quite simple. The environment has two methods, namely, *reset* and *step*.

The reset method sets the simulator into its initial state $s_0 \in \mathcal{S}$ and returns this state from the method. The step method performs a time step t of the environment, the method expects the action as its input. The agent provides the action based on the state provided by the environment in the previous step. The environment simulates a single time step and then returns a triplet, specifically the next state, reward and a boolean determining whether the state is terminal or not.

We now describe the internal dynamics of the simulation environment. There are two stochastic elements in the simulation. The first one is the

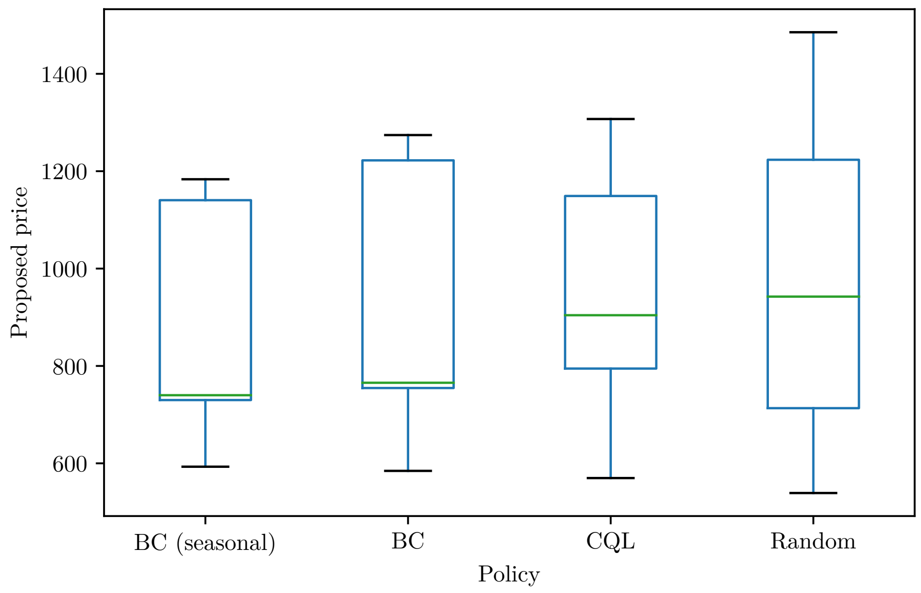


Figure 6.4: Distribution of proposed prices for each policy from the evaluation dataset (the green line represents the median)

customer arrival rate, we model it with a Poisson process. Specifically, we use $\lambda = 60$. We consider every tenth customer to be a buyer. The second stochastic element is the acceptable price the customer is willing to pay. We model the acceptable price as a uniform distribution with the range of $[500, 1500]$.

We always start from the initial state $s_0 \in \mathcal{S}$. Furthermore, we know that every state $s \in \mathcal{S}$ is an element of $\mathbb{R}^8$, where $\mathbb{R}^8$ represents a vector space with dimensions representing the features described in Section 5.1. We fix the following parameters based on the averages given in the historical data and suggestions of the domain expert. The initial state s_0 always begins in the same month, March. The product's stock price is 600 CZK. The initial selling price is 1231 CZK (this corresponds to 1490 CZK with VAT). The initial stock is 50 shirts. The simulation keeps the average time spent feature constant for more simplicity.

The general description of a simulation step is the following:

1. Receive the action from the agent as an input (the action is the new price).
2. Sample customers from the Poisson process, every tenth customer is

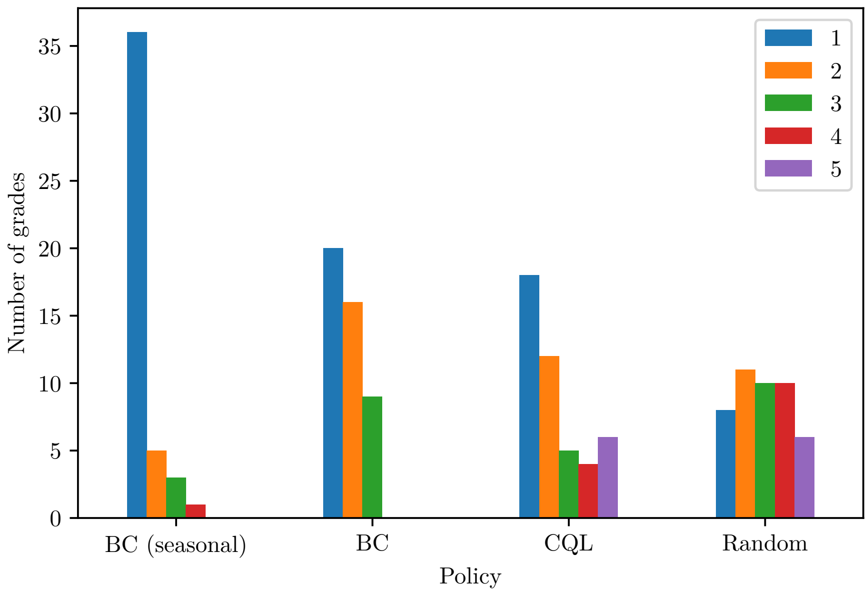


Figure 6.5: Histogram of grades for every policy in the expert evaluation

considered a buyer.

3. Take every buyer and sample their acceptable price from the uniform distribution (done for every buyer independently).
4. Calculate the number of bought shirts from the customer's acceptable price and the new price.
5. Update the state according to the number of bought products.
6. Return the next state, reward and boolean indication whether the state is terminal or not.

The simulation usually has $T = 12$ steps, where each step corresponds to one month. Sometimes, the stock can sell out faster and the simulation has fewer steps. This determines whether a state is terminal or not.

Primary Simulation Evaluation Metric

The primary metric we use for the simulation evaluation is the *average return*. The average return is the expected value of the cumulative reward when following a policy π. It is important to note that the policy π is fixed, we are evaluating the policy that was trained before. We give the exact formula for the average return in Equation 6.1. Notice that this equation does not include a discount factor γ, the reason for that is that we use the average return for the evaluation of a fixed policy. In contrast to this, we have the definition of the utility function in 4.1, where we use a discount factor to make early rewards more valuable (useful when training a policy).

$$R(\pi) = \mathbb{E}\left[\sum_{t=0}^{T} \mathcal{R}(s_t, a_t)\right] \tag{6.1}$$

We experimented with a lot of different representations of the reward function. First, we tried to use the revenue divided by unique page views as the reward function. This setting failed because the agent started to aggressively underprice to obtain rewards. We decided to switch to a reward function using profit, as described in Section 5.1. The results we obtained indicated success.

Baseline Policies

We need to introduce baseline policies against that we compare our trained algorithms. Our first baseline is a random policy. It seems reasonable to expect better results than those coming from a random policy. The implementation is rather simple, we use a uniform distribution over the action space to sample random actions.

The second baseline is a rule-based static policy. This policy corresponds to the current discounting strategy used by the e-commerce business. For the first three months, the business keeps the price at its starting price. After that, for the next two months, the business discounts the product by 20%. At last, the discount of 40% from the original price happens. We are trying to create a policy that can outperform the current strategy, so it makes sense to use the static policy as a baseline.

Simulation Results

We trained two algorithms, namely, CQL and BC, using the training dataset we describe in Section 3.2. Specifically, we trained the CQL algorithm for

2000 epochs and the BC algorithm for 50 epochs. It is important to realize the difference between CQL and BC. While CQL considers the rewards of the MDP, BC just mimics the training data. In our simple simulation, BC achieves better results than CQL, but we cannot expect BC to generalize well. This also explains the necessary difference in training epochs, learning a policy using rewards is much more difficult than just doing regular supervised learning.

We provide our simulation results in Figure 6.6. We expect CQL to outperform BC on data that requires generalization with respect to unseen states. For good generalization properties, we would need a large and most importantly diverse dataset. We back this up by an experiment in the next section.

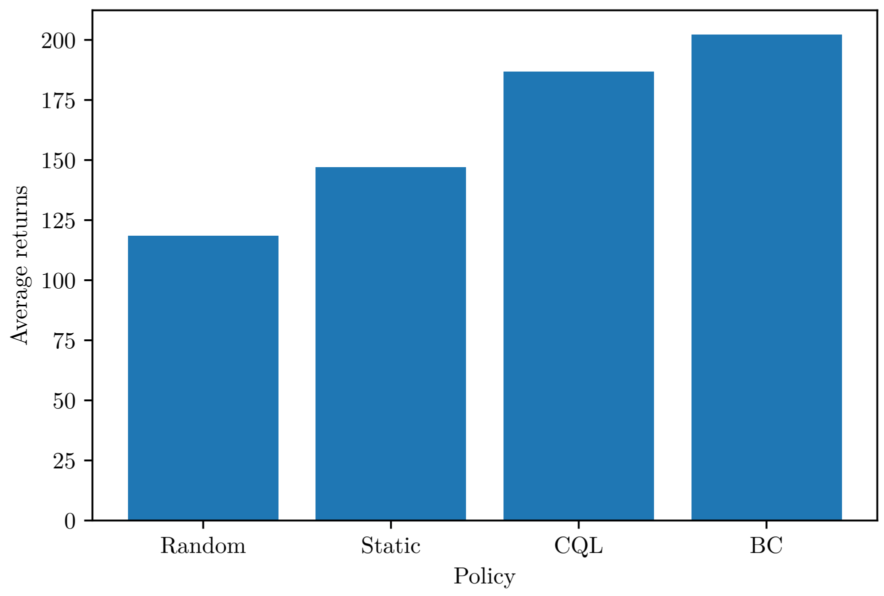


Figure 6.6: Simulation performance averaged over 100,000 runs

CQL and Data Diversity Experiment

While BC outperformed CQL in our simple simulation, we claim that CQL can perform better if we give it more data that is diverse. We validate our claim with a simple experiment:

First, we note that the baseline random policy performed worse than the baseline static policy (see Figure 6.6). Continuing with our experiment, we run both of these policies on our simulator and sample transitions from them. We sample around 1,000,000 MDP transitions from both of these policies. Next, we create offline datasets from the sampled transitions. We train the CQL algorithm using these offline datasets.

The results show that data diversity is a key factor in CQL training. Specifically, we know that the static policy only uses 3 discrete actions. On the other hand, the random policy uses actions from the whole action space. We present the results of this experiment in Figure 6.7. We can see that CQL learns an optimal policy from the random policy dataset and fails when trained on the static policy dataset.

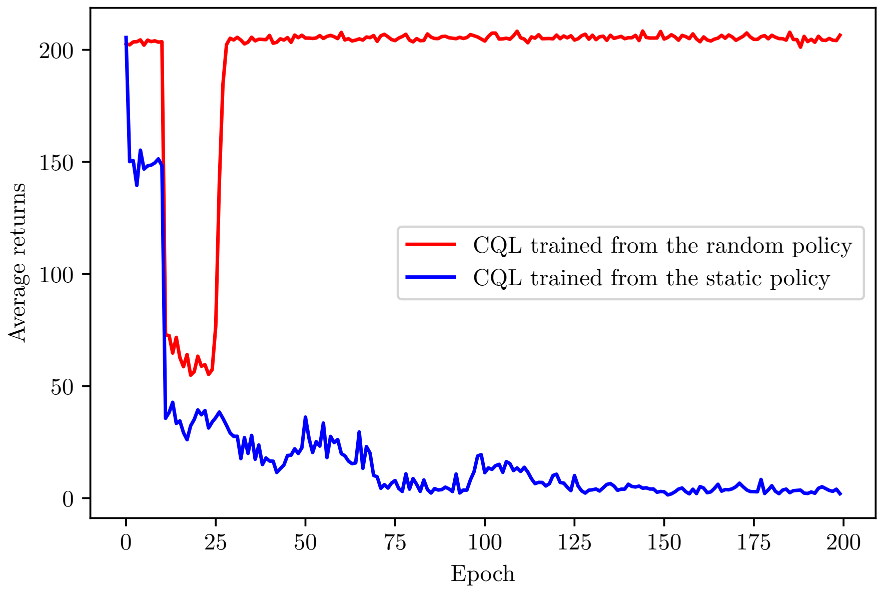


Figure 6.7: Training of CQL on sampled transitions from the random and static policies from the simulator

Possible future work with a larger and more diverse dataset could enable us to train a better CQL policy with strong generalization properties. The latter cannot be said about BC, because BC does not consider the reward function and therefore is not able to generalize.

Chapter 7

Conclusions

In this work, we investigated the problem of learning dynamic pricing policies from historical datasets in the setting of e-commerce, specifically fashion retail.

We chose to approach the problem of dynamic pricing via the methods of Offline RL and IL. We used Offline RL and IL because these methods can learn from offline historical data without the need of a simulator. To obtain this historical data, we created a data pipeline that retrieves a dataset from a real-world e-commerce store, specifically it retrieves the data from two separate services, the first one is the administration interface, and the second one is the GA API. The most challenging part was the data cleaning and merging into the final dataset.

While the real-world dataset is not perfect, we still managed to successfully train policies based on CQL and BC algorithms and achieve great results. Specifically, we carried out two separate evaluations, the first one, an evaluation by a domain expert, showed that the best method, BC, achieved an average grade of 1.31 out of 5. With 1 being the best possible score, we obtained an almost perfect score, furthermore, all other methods we proposed outperformed the random benchmark. The second evaluation done in a simulator showed that all our proposed methods outperformed the static baseline method by a minimum margin of 27%.

We consider these results a success, since this is the first application of Offline RL to dynamic pricing problems.

The main weakness of this work is the small dataset. This is indeed related to the nature of a real-world application, but we admit that the dataset is the major bottleneck in the training of policies.

To summarize, in this work, we have for the first time applied Offline RL methods to dynamic pricing. For this application, we have collected a new dataset which we used to train multiple pricing policies. For evaluation, we used two fundamentally different approaches: a domain expert and a simulation. Both methods have shown the viability of Offline RL for dynamic pricing.

Future work

As the scope of this work is large, there are a lot of possibilities for future work. One possible future work lies in the choice of the algorithm itself, the research in the domain of RL is very active, and moving forward quickly, new methods could enable us to train even better policies that generalize well.

The model itself is a possible direction of future work, specifically, the evaluation of possible features and state space choices, a comparison of discrete and continuous action spaces, or an analysis of the possible reward functions are all promising directions that could improve the pricing solution.

The improvement of evaluation methods is also a promising idea for future work, specifically, a more sophisticated simulation or any novel method for offline evaluation could greatly improve the whole dynamic pricing process.

Glossary

A3C Asynchronous Advantage Actor-Critic. 10, 28

BC Behavioral Cloning. 5, 12, 19, 35, 36, 44, 45, 49–51, 53

BCQ Batch-Constrained Deep Q-learning. 11, 33

BEAR Bootstrapping Error Accumulation Reduction. 11, 33

CQL Conservative Q-learning. 5, 10, 11, 19, 32–36, 44, 45, 49–51, 53

CSV Comma-separated values. 15, 16

DDPG Deep Deterministic Policy Gradient. 10, 11

DDPGfD Deep Deterministic Policy Gradient from Demonstrations. 9, 11

DQfD Deep Q-learning from Demonstrations. 9, 11

DQN Deep Q-network. 10, 11, 24–26, 30, 31

DRL Deep Reinforcement Learning. 9, 10, 23, 25, 26, 33

GA Google Analytics. 14–16, 38, 39, 53

IL Imitation Learning. 5, 10, 12, 19, 35, 36, 53

MDP Markov Decision Process. 5, 9, 19–21, 35–37, 40, 50, 51

Replay Buffer is an array of chosen size that stores prior transitions that have been observed in the MDP. It is used to break the correlation between new transitions coming from a similar policy. For more information, see [28]. 10, 11, 25, 31, 32, 34, 35

RL Reinforcement Learning. 1, 5, 7–12, 19, 21–27, 29, 31, 33–36, 53, 54

SAC Soft Actor-Critic. 10, 11, 19, 26, 27, 30–34

SGD Stochastic Gradient Descent. 25, 28, 30–32, 36

TD3 Twin Delayed Deep Deterministic Policy Gradient. 30

VAE Variational Auto-Encoder. 11

Bibliography

[1] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. `https://gym.openai.com`, 2016.

[3] Felipe Caro and Jérémie Gallien. Clearance Pricing Optimization for a Fast-Fashion Retailer. *Operations Research*, 60(6):1404–1422, December 2012.

[4] Yan Cheng. Dynamic Pricing Decision for Perishable Goods: A Q-Learning Approach. 2008.

[5] Yan Cheng. Real Time Demand Learning-Based Q-learning Approach for Dynamic Pricing in E-retailing Setting. In *2009 International Symposium on Information Engineering and Electronic Commerce*, pages 594–598, Ternopil, Ukraine, 2009. IEEE.

[6] Arnoud V. den Boer. Dynamic pricing and learning: Historical origins, current research, and new directions. *Surveys in Operations Research and Management Science*, 20(1):1–18, June 2015.

[7] Joan Morris DiMicco, Amy Greenwald, and Pattie Maes. Dynamic pricing strategies under a finite time horizon. In *Proceedings of the 3rd ACM conference on Electronic Commerce - EC '01*, pages 95–104, Tampa, Florida, USA, 2001. ACM Press.

[8] Kris Johnson Ferreira, Bin Hong Alex Lee, and David Simchi-Levi. Analytics for an Online Retailer: Demand Forecasting and Price Optimization. *Manufacturing & Service Operations Management*, 18(1):69–88, February 2016.

[9] Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4RL: Datasets for Deep Data-Driven Reinforcement Learning. *arXiv:2004.07219 [cs, stat]*, February 2021. arXiv: 2004.07219.

[10] Scott Fujimoto, David Meger, and Doina Precup. Off-Policy Deep Reinforcement Learning without Exploration. *arXiv:1812.02900 [cs, stat]*, August 2019. arXiv: 1812.02900.

[11] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. *arXiv:1802.09477 [cs, stat]*, October 2018. arXiv: 1802.09477.

[12] The garage contributors. Garage: A toolkit for reproducible reinforcement learning research. `https://github.com/rlworkgroup/garage`, 2019.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[14] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement Learning with Deep Energy-Based Policies. *arXiv:1702.08165 [cs]*, July 2017. arXiv: 1702.08165.

[15] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv:1801.01290 [cs, stat]*, August 2018. arXiv: 1801.01290.

[16] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic Algorithms and Applications. *arXiv:1812.05905 [cs, stat]*, January 2019. arXiv: 1812.05905.

[17] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep Q-learning from Demonstrations. *arXiv:1704.03732 [cs]*, November 2017. arXiv: 1704.03732.

[18] Jonathan Ho, Jayesh K. Gupta, and Stefano Ermon. Model-Free Imitation Learning with Policy Optimization. *arXiv:1605.08478 [cs]*, May 2016. arXiv: 1605.08478.

[19] W. Jintian and Z. Lei. Application of reinforcement learning in dynamic pricing algorithms. In *2009 IEEE International Conference on Automation and Logistics*, pages 419–423, August 2009. ISSN: 2161-816X.

[20] Jeffrey O. Kephart, James E. Hanson, and Amy R. Greenwald. Dynamic pricing by software agents. *Computer Networks*, 32(6):731–752, May 2000.

[21] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, January 2017. arXiv: 1412.6980.

[22] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. *arXiv:1312.6114 [cs, stat]*, May 2014. arXiv: 1312.6114.

[23] Aviral Kumar, Justin Fu, George Tucker, and Sergey Levine. Stabilizing Off-Policy Q-Learning via Bootstrapping Error Reduction. *arXiv:1906.00949 [cs, stat]*, November 2019. arXiv: 1906.00949.

[24] Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative Q-Learning for Offline Reinforcement Learning. 2020.

[25] Sascha Lange, Thomas Gabel, and Martin Riedmiller. Batch Reinforcement Learning. In Marco Wiering and Martijn van Otterlo, editors, *Reinforcement Learning*, volume 12, pages 45–73. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. Series Title: Adaptation, Learning, and Optimization.

[26] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. *arXiv:2005.01643 [cs, stat]*, November 2020. arXiv: 2005.01643.

[27] 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:1509.02971 [cs, stat]*, July 2019. arXiv: 1509.02971.

[28] Long-Ji Lin. Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *REINFORCEMENT LEARNING*, 1992.

[29] Jiaxi Liu, Yidong Zhang, Xiaoqing Wang, Yuming Deng, and Xingyu Wu. Dynamic Pricing on E-commerce Platform with Deep Reinforcement Learning. *arXiv:1912.02572 [cs, stat]*, December 2019. arXiv: 1912.02572.

[30] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. 2016.

[31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. 2013.

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[33] Y. Narahari, C. V. L. Raju, K. Ravikumar, and Sourabh Shah. Dynamic pricing models for electronic business. *Sadhana*, pages 2–3, 2005.

[34] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J. Andrew Bagnell, Pieter Abbeel, and Jan Peters. An Algorithmic Perspective on Imitation Learning. *Foundations and Trends in Robotics*, 7(1-2):1–179, 2018.

[35] C. V. L. Raju, Y. Narahari, and K. Ravikumar. Learning dynamic prices in electronic retail markets with customer segmentation. *Annals of Operations Research*, 143(1):59–75, March 2006.

[36] C.V.L. Raju, Y. Narahari, and K. Ravikumar. Reinforcement learning applications in dynamic pricing of retail markets. In *IEEE International Conference on E-Commerce, 2003. CEC 2003.*, pages 339–346, Newport Beach, CA, USA, 2003. IEEE Comput. Soc.

[37] Rupal Rana and Fernando S. Oliveira. Dynamic pricing policies for interdependent perishable products or services using reinforcement learning. *Expert Systems with Applications*, 42(1):426–436, January 2015.

[38] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial intelligence: a modern approach.* Prentice Hall series in artificial intelligence. Prentice Hall, Upper Saddle River, 3rd ed edition, 2010.

[39] Daniel Seita. Offline Reinforcement Learning: How Conservative Algorithms Can Enable New Applications. `http://bair.berkeley.edu/blog/2020/12/07/offline/`, 2020.

[40] Manu Sridharan and Gerald Tesauro. Multi-agent Q-learning and regression trees for automated pricing decisions. 2002.

[41] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction.* Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018.

[42] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. 1999.

[43] Gerald Tesauro. Pricing in agent economies using neural networks and multi-agent Q-learning. In *Lecture Notes in Computer Science*, pages 288–307. Springer-Verlag, 1999.

[44] Gerald J. Tesauro and Jeffrey O. Kephart. Foresight-based pricing algorithms in an economy of software agents. In *Proceedings of the first international conference on Information and computation economies - ICE '98*, pages 37–44, Charleston, South Carolina, United States, 1998. ACM Press.

[45] Mel Vecerik, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin Riedmiller. Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards. *arXiv:1707.08817 [cs]*, October 2018. arXiv: 1707.08817.

[46] David Vengerov. A gradient-based reinforcement learning approach to dynamic pricing in partially-observable environments. *Future Generation Computer Systems*, 24(7):687–693, July 2008.

[47] C. J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, Oxford, 1989.