UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Dynamic Pricing using Reinforcement Learning for the Amazon marketplace

Author:
Andrii PRYSIAZHNYK

Supervisor:
PhD Taras FIRMAN

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2021

Declaration of Authorship

I, Andrii PRYSIAZHNYK, declare that this thesis titled, "Dynamic Pricing using Reinforcement Learning for the Amazon marketplace" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Dynamic Pricing using Reinforcement Learning for the Amazon marketplace

by Andrii PRYSIAZHNYK

Abstract

This thesis proposes and compares a few approaches for tackling the dynamic pricing problem for e-commerce platforms. Dynamic pricing engines may help e-retailers to increase their performance indicators and gain useful market insights. We worked with the Amazon marketplace, using customer sales data along with additional data from the Amazon services. Demand forecasting-based and RL-based pricing strategies were considered. We gave a detailed explanation of each method, commenting on its pros and cons. In order to train RL agents and compare them with baseline methods, the simulator of the market environment was built. Conducted experiments proved the effectiveness and advantages of RL-based methods over the classic approaches. We also propose the idea for future works on how RL-based pricing could be further enhanced. The source code of our study is publicly available on GitHub.

Acknowledgements

First of all, I want to thank my Mother and Grandparents for my upbringing and education. Also, I am very grateful to Eleks DSO team, especially to Taras Firman and Volodymyr Getmanskyi, for their trust, constant help, and the opportunity of doing exciting things. Finally, I want to thank Ukrainian Catholic University for the great professors and students that surround me over the last four years.

Contents

List of Figures

List of Tables

List of Abbreviations

RL	**R**einforcement **L**earning
ML	**M**achine **L**earning
DS	**D**ata **S**cience
MDP	**M**arkov **D**ecision **P**rocess
DP	**D**ynamic **P**rogramming
GPI	**G**eneralized **P**olicy **I**teration
MC	**M**onte **C**arlo
TD	**T**emporal **D**ifference
ASIN	**A**mazon **S**tandard **I**dentification **N**umber
EDA	**E**xploratory **D**ata **A**nalysis
ARMA	**A**uto**R**egressive **M**oving **A**verage
ARIMA	**A**uto**R**egressive **I**ntegrated **M**oving **A**verage
SARIMA	**S**easonal **A**uto**R**egressive **I**ntegrated **M**oving **A**verage
MAPE	**M**ean **A**bsolute **P**ercentage **E**rror
WMAPE	**W**eighted **M**ean **A**bsolute **P**ercentage **E**rror
RMSE	**R**oot **M**ean **S**quare **E**rror
GLM	**G**eneralized **L**inear **M**odel
ACF	**A**uto**c**orrelation **F**unction
PACF	**P**artial **A**uto**c**orrelation **F**unction
DQN	**D**eep **Q**-**N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
RNN	**R**ecurrent **N**eural **N**etworks
LSTM	**L**ong **S**hort **T**erm **M**emory
GRU	**G**ated **R**ecurrent **U**nit
ANN	**A**rtificial **N**eural **N**etwork
RBF	**R**adial **B**asis **F**unction
DDPG	**D**eep **D**eterministic **P**olicy **G**radient
RAM	**R**andom **A**ccess **M**emory
PPC	**P**ay **P**er **C**lick

Dedicated to my Grandfather

Chapter 1

Introduction

Dynamic pricing is the task of finding an optimal pricing strategy. The company integrates the dynamic pricing engines to increase different performance indicators, such as income or revenue. A dynamic pricing engine considers many factors, such as the company's previous prices and sales, competitors' prices, spend on advertising, seasonality, and other exogenous factors, to find the optimal strategy. Dynamic pricing algorithms are used in many businesses such as retail, airlines, ride-hailing companies, and rental companies to effectively respond to market fluctuations. For instance, Uber pushes prices up in case of high demand caused by a particular time range or bad weather.

The intuition behind the dynamic pricing is the following. At each timestep, the demand curve could have different forms. This is because demand, except price, also depends on exogenous factors that are changing in time. This means that at different timesteps, it is beneficial for a company to set different prices. Dynamic pricing engines are trying to "understand" the demand curve at the current timestep and, optionally, how our pricing strategy affects demand curves in the future.

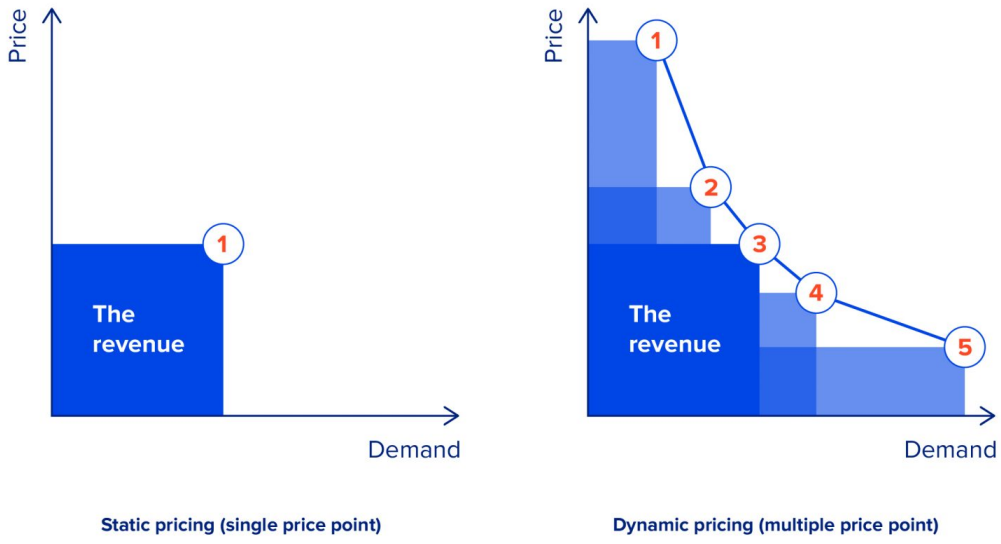


FIGURE 1.1: Static pricing vs Dynamic pricing. Image is taken from [2].

1.1 Dynamic pricing for e-commerce

In this work, we will consider dynamic pricing for e-commerce platforms. More specifically, we will build and compare a few dynamic pricing engines for the Amazon marketplace. The methods described in this work are general enough and could be effectively applied to other domains. The reason for working with the Amazon marketplace is the rapid development of e-commerce, the business value of dynamic pricing for e-retailers, and the availability of sales data. Also, there are online services, such as Keepa, that allow buying additional data from Amazon. This data could significantly boost modeling. An enormous number of factors influence current sales. Some factors are common to different marketplaces, such as seasonality, competitors' prices, or the company's previous sales. On the Amazon marketplace, there are additional factors such as sales-rank and spend on PPC advertising. Sales-rank indicator shows how much the product is sold compared to other products from the same category. The PPC advertising model lets sellers increase their sales by investing money in the paid advertising. All those factors could be integrated into a dynamic pricing engine in order to enhance its performance.

Nowadays, the e-commerce sector is one of the "big players" on the world market. In 2019, e-commerce generated around $3.53 trillion in revenue. Moreover, there are expectations that by 2040 around 95% of all purchases will be made online. E-commerce offers convenience and personalization to its customers, which other shopping methods cannot fulfill. Moreover, while the pandemic blockades the majority of industries, e-commerce only benefits from the quarantine measures and the lack of physical interactions. We could see this from the following figures:

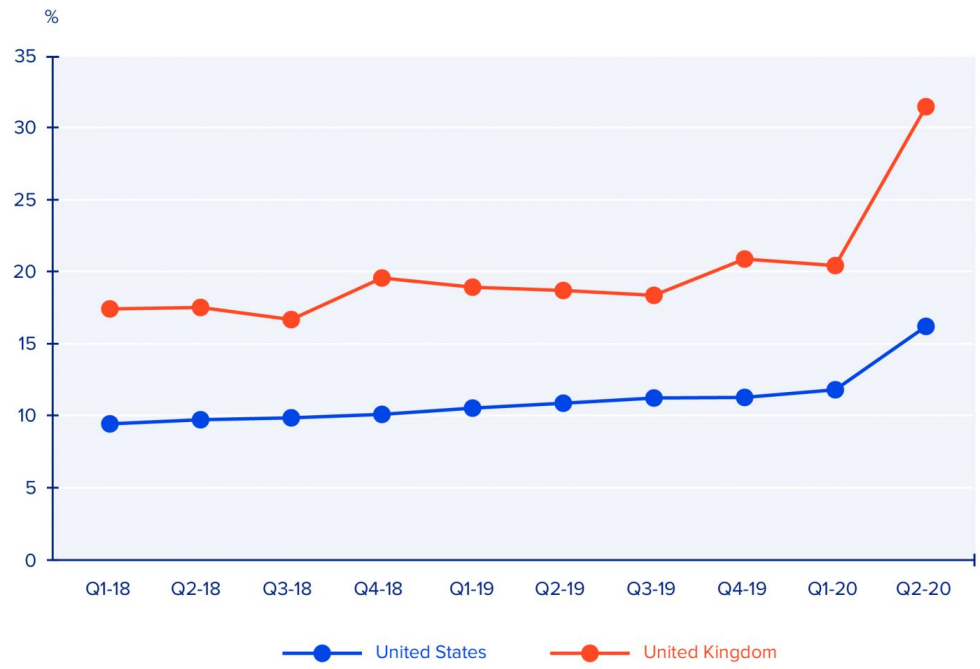


FIGURE 1.2: Pandemic influence on the e-commerce share. Image is taken from [2].

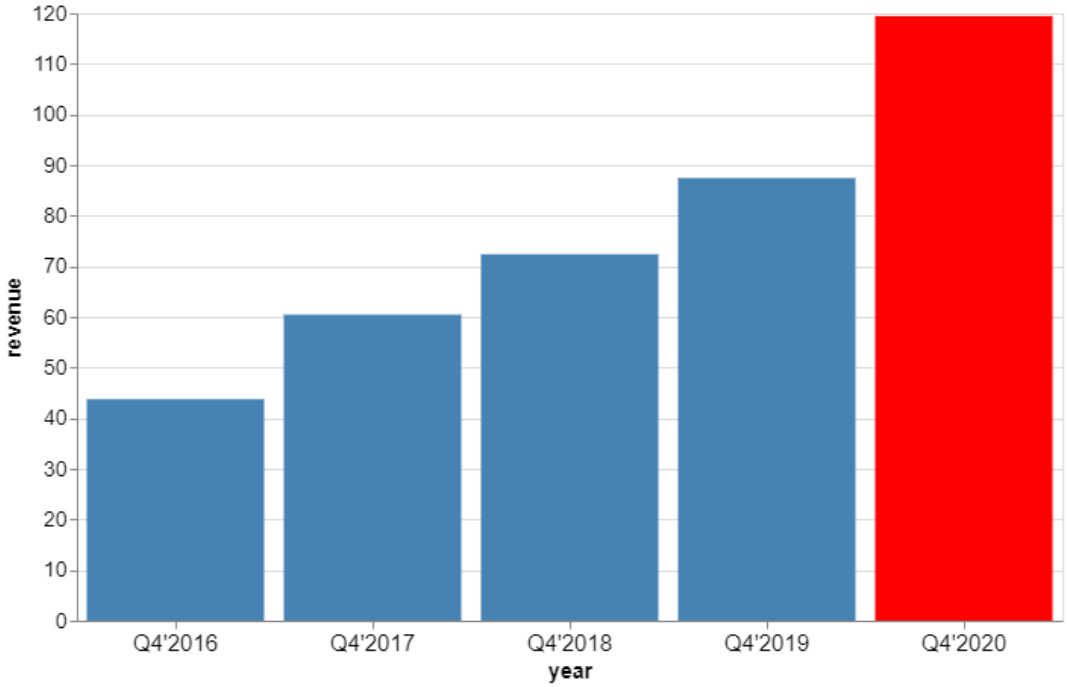


FIGURE 1.3: Pandemic influence on the Amazon revenue.

For e-retailers, it is crucial to monitor the prices and the market state. Overpricing of products may result in the deterrence of the possible customers. On the other hand, undercharging may limit the company's income. For example, after implementing dynamic pricing, Best Buy increased its sales by 25%. The main advantages of dynamic pricing for e-commerce retailers are the following:

1. Stay ahead of competitors. Dynamic pricing automates the process of monitoring competitors' prices. This allows attracting new customers and increasing customer loyalty.

2. Increase in profits. This is a priority task of dynamic pricing. As was stated earlier, any other economic performance indicator may be used as an objective.

3. Gain market insights. Interpretability of ML and DS solutions is a hot topic nowadays. Model interpretability may be a crucial factor before deploying it to the production-ready system. Interpreting and understanding outputs of dynamic pricing engines enable companies to stay aware of current market trends. Also, this allows the company to understand how its customers behave.

1.2 Thesis structure

In Chapter 2, the technical background needed to understand implemented modeling steps is described. It comprehensively covers RL basics such as Markov Decision Process, Bellman optimality equation, Dynamic Programming methods, Monte Carlo methods, and Temporal Difference learning. In Chapter 3, we describe related works on the dynamic pricing topic, dividing dynamic pricing engines into two major categories - demand forecasting based and RL based. In Chapter 4, the Amazon

data, which is used for modeling, is presented, and the process of building a demand forecasting model is described. Also, we explain in more detail two types of dynamic pricing strategies along with different RL methods, which we implement and compare in the thesis. In Chapter 5, the experimental results of training RL agents and comparing their performance with baseline methods (demand forecasting based strategy and random strategy) are presented. Finally, in Chapter 6, we end up with conclusions, commenting on the pros and cons of different methods.

Chapter 2

Technical background

RL basics are written mainly based on "Reinforcement Learning: An introduction" book [27].

2.1 Introduction to RL

RL is one of three main areas of ML, alongside supervised learning and unsupervised learning. Its primary task is to develop algorithms that allow agents to maximize their cumulative rewards while acting in an unknown environment. RL algorithms are concerned with learning what to do - how to map situations to actions.

The idea of learning by interaction with an unknown environment is very natural and, in some way, is biologically inspired. As human beings, we continuously interact with the world around us, acting in a way to maximize our reward, for example, satisfaction, acknowledgment, or money. RL is an exceptional modeling framework in the sense that it is trying to mimic human or animal decision-making algorithms.

RL significantly differs from other types of learning. Supervised learning is concerned with building models based on a training set of input-output pairs. It is vital that models could generalize to unseen data and have enough capacity to capture all regular dependencies between dependent and target variables. A problem of supervised learning is to find a rule that maps elements from the input space into output space elements based on a small subset of correct mappings. RL could not be deduced to a supervised learning setting. The reason is the following: it is impractical to get a subset of optimal behavior samples, which is both correct and representative of all situations in which an agent could be. RL differs from unsupervised learning in a way that it involves maximization objectives. Despite this, both supervised learning and unsupervised learning algorithms could be used to boost the performance of RL algorithms. Unsupervised learning could be used to discover the hidden structure of state-space and end up with a more compact representation. Supervised learning models could be used to approximate optimal action-values or policy (see sections on DQN [9] and Policy Gradients [28]).

Exploration-exploitation tradeoff is a feature of RL that did not appear in other ML areas. The dilemma is whether an agent should exploit behavior that turned out to be good in the past or explore new actions that could possibly lead to even greater rewards. Almost all RL algorithms should integrate this exploration mechanism and balance between exploration and exploitation. Exploration is usually decreased with time. In the beginning, the agent needs to explore more. In the case of nonstationarity, the exploration rate could not be reduced up to zero because the environment continually changes its behavior.

Because of its generality, RL can be used to tackle a great number of problems from different domains. The only required thing is an environment (it should possess specific properties, see the section on MDP). That is way, RL is tightly coupled with other engineering and scientific disciplines. Also, RL strongly interacts with psychology and neuroscience [32]. Lots of RL algorithms were biologically inspired. On the other hand, RL provides a model of animal learning that fits empirical data pretty well. The link between these disciplines is due to the similarity of RL to what humans and animals do.

RL is used successfully in trading [29] and finance [16] (dynamic pricing), NLP (text generation [33], question answering [22], text summarization [11]), healthcare (dynamic treatment regimes [18]), engineering (cluster resource management [19]), robotics (robotics grasping [10]), gaming (AlphaGo [25]).

2.2 Markov Decision Process (MDP)

This section will define the fundamental notions of RL, such as environment, agent-environment interaction, policy, returns, state-value function, and action-value function.

The environment is usually formalized with MDP. MDP is a discrete-time stochastic control process. It provides a mathematical framework for modeling sequential decision making, in which an agent's actions may influence future outcomes. For now, we will restrict ourselves to finite MDPs. It consists of a state-space S, action-space A, reward-space R, and dynamics function p. Sets S, A, and R are all finite. Function p describes the dynamics of the MDP.

$$p(s', r, s, a) = P(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a) \tag{2.1}$$

$$\sum_{s' \in S} \sum_{r \in R} p(s', r, s, a) = 1$$

A significant restriction put on MDP is the so-called Markov property. It states that the future is independent of the past, given the present. In terms of MDP, Markov property means that given S_t and A_t, R_{t+1} and S_{t+1} are independent with R_k and S_k for all $k < t$. Markov property is rather a restriction on states, which should compress enough information about the past.

$P(S_{t+1} = s', R_{t+1} = r | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}, ...) = P(S_{t+1} = s', R_{t+1} = r | S_t = s_s, A_t = a_t)$

More specifically, agent and MDP interact at each timestep $t = 0, 1, 2, \dots$. The agent takes action a, based on the current state s. MDP receives a and responds with reward r and next state s'. Together they produce the following sequence of elements

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$$

The overall process is visualized in the Figure 2.1.

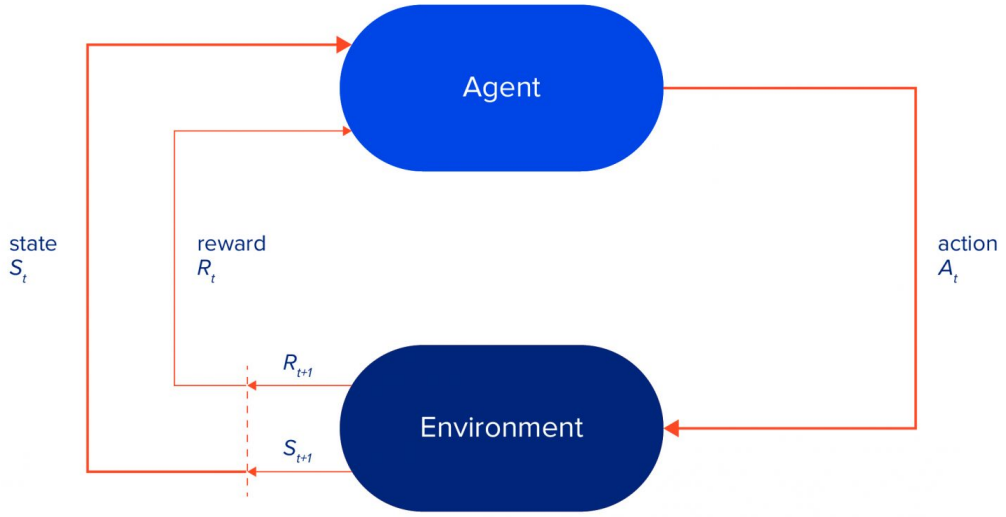


FIGURE 2.1: Agent-Environment interface. Image is taken from [2].

The learning objective is to maximize the cumulative reward the agent receives in the long run while acting in an unknown MDP. It is also called a reward hypothesis. To formalize this idea, the notion of return should be defined. The return is some function of a sequence of rewards. In the case of episodic tasks (those that have a finite number of interactions with an environment), it can be defined as

$$G_t = R_{t+1} + R_{t+1} + R_{t+2} + ... + R_T$$

In the case of continuing tasks (those that have an infinite number of interactions with an environment), we should also consider convergence properties of the infinite sums. That is why the following return is used,

$$G_t = R_{t+1} + \gamma \cdot R_{t+1} + \gamma^2 \cdot R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where γ is a discount factor, which lies in a unit interval and controls how much an agent is concerned with the future rewards. For example, if γ is equal to 0, return at timestep t is equal to the immediate reward at the corresponding timestep. Then in order to maximize the expected return from each state, the agent could use a greedy strategy, maximizing the expected immediate reward. This could yield a significant reduction of cumulative reward and thus is suboptimal in the long run.

The policy is a mapping from state-space to probabilities of selecting each possible action. Informally, the policy defines the strategy which agent uses for moving on MDP. RL algorithms are trying to find the best policy out of the whole space of possible policies.

$$\pi(s|a) = P(A_t = a|A_t = s) \tag{2.2}$$

In order to define an optimal policy, the notion of state-value and action-value functions should be defined. Those functions could be calculated for any policy, and, roughly speaking, they show how good a policy is in different parts of a state-space.

The state-value function is defined on a state-space and shows an expected return an agent will receive in some state under a given policy. Similarly, the action-value function is defined on a cartesian product of state-space and action-space. It shows an expected return under a given policy in case of taking a specific action from a given state.

$$v_\pi(s) = E_\pi[G_t|S_t = s] \tag{2.3}$$

$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a] \tag{2.4}$$

A crucial equation in RL that linearly relates different state-values is the Bellman equation. They significantly simplify finding state-values under a given policy either with known or unknown dynamics function. First, we could relate successive returns with the following formula

$$\begin{aligned} G_t &= R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= R_{t+1} + \gamma \cdot (R_{t+2} + \gamma \cdot R_{t+3} + ...) \\ &= R_{t+1} + \gamma \cdot G_{t+1} \end{aligned} \tag{2.5}$$

This equation could be used to derive the Bellman equation for v

$$\begin{aligned} v_\pi(s) &= E_\pi(G_t|S_t = s) \\ &= E_\pi[R_{t+1} + \gamma \cdot G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r,s,a) \cdot (r + \gamma \cdot E_\pi[G_{t+1}|S_{t+1} = s']) \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r,s,a) \cdot (r + \gamma \cdot v_\pi(s')) \end{aligned} \tag{2.6}$$

If the model of the environment is known, then one could easily find a state-value function by solving a system of linear equations. It has been proved that the following system has a unique solution. If the model is unknown, lots of algorithms that estimate state-values from the interaction with an environment exploit Bellman equations.

An important result in RL, which is a basis for all action-value methods, is called a Policy Improvement Theorem. Later we will show how it allows us to iteratively improve our policies until we get an optimal one. It states that if we have two policies that satisfy the following set of inequalities

$$(\forall s \in S) \sum_a \pi'(a|s) \cdot q_\pi(s,a) \geq v_\pi(s)$$

then one can show that

$$(\forall S \in S) v_{\pi'}(s) \geq v_\pi(s) \tag{2.7}$$

In this case, we say that policy π' is better than policy π. It is a very intuitive definition since, in any state, it is at least as good to follow policy π' than policy π.

In particular, if there are two deterministic policies, such that the first one is greedy with respect to the action-value function of the second one then clearly the greedy policy is better.

$$q_\pi(s, \pi'(s)) \geq v_\pi(s)$$

It can be shown that there exists at least one policy that is better than any other policy. Its state-values and action-values should satisfy the following equations - Bellman optimality equations for v and q (can be proved with Policy Improvement Theorem):

$$\begin{aligned} v_*(s) &= \max_a q_{\pi_*}(s,a) \\ &= \max_a E_{\pi_*}[G_t | S_t = s, A_t = a] \\ &= \max_a E_{\pi_*}[R_{t+1} + \gamma \cdot G_{t+1} | S_t = s, A_t = a] \\ &= \max_a E[R_{t+1} + \gamma \cdot v_{\pi_*}(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',a} p(s', r, s, a) \cdot (r + \gamma \cdot v_*(s')) \end{aligned} \tag{2.8}$$

$$\begin{aligned} q_*(s,a) &= E[R_{t+1} + \gamma \cdot \max_{a'} q_*(S_{t+1}, a) | S_t = s, A_t = a] \\ &= \sum_{s',a} p(s', r, s, a) \cdot (r + \gamma \cdot \max_{a'} q_*(s', a')) \end{aligned} \tag{2.9}$$

We will call this policy an optimal policy. Moreover, any policy, which is greedy with respect to optimal action-values is optimal. This fact is exploited by all action-value methods, which aim to approximate optimal action values. Solve MDP means finding an optimal policy. The easiest way to solve the MDP is to solve the Bellman optimality equation.

2.3 Dynamic Programming (DP)

DP algorithms find optimal policy given MDP with known dynamics function. In the real world, the assumption of known dynamics functions is usually violated. Still, it is worth considering these algorithms because of their theoretical importance. All methods considered later will use the same general strategy for finding optimal policy without assuming a perfect model of the environment. The idea is straightforward and naturally arises from the Policy Improvement Theorem. Start with any deterministic policy and continually update it to be greedy to estimated action-values until convergence.

The first step required to implement this idea is to estimate the state-values of a given policy. The simplest way to find state-values is to solve a system of linear equations (Bellman equations, the existence, and uniqueness is guaranteed under some technical assumptions). This approach is computationally expensive. One can use the following update rule, which in the limit converges to the true solution.

$$\begin{aligned} v_{k+1}(s) &= E_\pi[R_{t+1} + \gamma \cdot v_k(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r, s, a) \cdot (r + \gamma \cdot v_k(s')) \end{aligned} \tag{2.10}$$

It is an iterative method to solve a system of linear equations. This algorithm is called policy evaluation. It terminates when state-value estimates change is small

enough. It is controlled with a special threshold, which is a hyperparameter of the algorithm. Having an efficient algorithm for estimating state-value function, we can define a policy iteration algorithm [23]. It starts with any policy and alters between two steps:

1. Estimate state-value function.
2. Change a policy so that it is greedy with respect to current action-values.

This process produces a sequence of policies and state-value functions

$$\pi_0 \longrightarrow v_{\pi_0} \longrightarrow \pi_1 \longrightarrow v_{\pi_1} \longrightarrow \pi_2 \longrightarrow v_{\pi_2} \longrightarrow ... \longrightarrow \pi_* \longrightarrow v_*$$

If the policy is no longer updated, then it is greedy with respect to its action-values. It means that the Bellman optimality equation is satisfied, and thus optimal policy is reached. Obviously, in policy iteration algorithms, we update our state-value estimates only a finite number of times. This speeds up computations and still possesses acceptable convergence properties.

In case we perform only one update during policy evaluation, we can explicitly write an update formula for state-values. This result is known as Value Iteration Algorithm [23].

$$\begin{aligned} v_{k+1}(s) &= \max_a E[R_{t+1} + \gamma \cdot v_k(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r, s, a) \cdot (r + \gamma \cdot v_k(s')) \end{aligned} \tag{2.11}$$

Another way of looking at this update formula is considering this update as an iterative algorithm for solving the Bellman optimality equation.

GPI is the general idea of letting the policy-evaluation and policy-improvement processes interact, independent of the granularity. Almost all RL algorithms are GPI-style. Even if we know the perfect model of the environment, it is usually impractical to find exact state-values. If state-values are estimated from experience (see the section on MC and TD methods), the true values are approached only in the limit.

2.4 Monte-Carlo methods (MC)

MC methods are used for estimating value functions and discovering optimal policies. These methods do not require a perfect model of the environment, rather experience - samples of state-action-reward sequences generated from the interaction with an environment. The idea is the following since state-values and action-values are expectations of some random variable (return function), they could be approximated with an average over realizations of this random variable. To ensure that well-defined returns are available, we will consider episodic tasks. MC methods have no natural extension to continuing tasks.

$$v_\pi(s) = E_\pi[G_t | S_t = s] \approx \frac{1}{n} \sum_{i=1}^{n} G_{ti} \tag{2.12}$$

First, we will focus on the estimation of state-value function. Suppose we are given episodes while following some fixed policy. To estimate a state's value, we

should find the entrance of this state and calculate respective returns. There are two standard options for doing this:

1. Use only those returns that correspond to the first entrance of a state in the episode - first-visit MC.
2. Use all entrances of a state in each episode - every-visit MC.

They have quite different theoretical properties, though both converge to true values as the number of entrances approaches infinity. Usually, first-visit MC is used.

Since dynamics function p is unknown, state-values are not enough to perform policy improvement steps. We should change a policy so that it is greedy to current action-value estimates. We could use the same first-visit MC or every-visit MC methods for estimation of action-value function. The problem is that after the policy improvement step, we'll get a deterministic policy. This means that we couldn't get estimates for actions, which are not greedy since they won't appear in any episode. This is actually an exploration vs exploitation trade-off dilemma. To get estimates and possibly discover better actions, we should explore, but exploitation of deterministic policy is a better choice for short-term reward maximization. This problem can be tackled by assuming exploring starts are available, using epsilon-soft policies or using off-policy methods.

Exploring starts [17] is a feature of the environment that allows it to be set in any state. With this feature, we could start our episodes for any state and action and thus could estimate action-values for any state-action pair. This assumption is kind of strong and usually is violated. Other approaches are more general and powerful.

Another option is to restrict a search space to a set of soft policies. To ensure that each action-value will receive a reliable estimate in the long run, we need each action to be taken with nonzero probability. Such a policy is called a soft policy. We can actually change the policy improvement step so that it produces a soft policy. For doing this, we will update our policy in a way that it uses a greedy action with probability 1 - ϵ and selects random action with probability ϵ. By Policy Improvement Theorem, each subsequent policy will be better than a previous one. Obviously, the sequence will not converge to the optimal policy, but it can be shown that the resulting policy will be the best out of all ϵ-soft policies.

The last option is using off-policy learning. It is dominated among action-value methods. The idea of off-policy learning is to use two policies, one that is learned about and becomes optimal in the limit, and another, more exploratory, used to generate experience. The policy learned about is called target policy, and policy used to generate experience - behavior. Most off-policy methods use the idea of Importance Sampling, which is the general technique for estimating the expected value of one distribution, given samples from another one. Let π and b be target and behavior policies, respectively. Then importance-sampling ratio is given by the following formula

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k, S_k) \cdot p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k, S_k) \cdot p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{2.13}$$

An important note here is that the importance-sampling ratio does not depend on the dynamics function, which is unknown. Then the state-value function of target policy can be calculated with the following formula

$$v_\pi(s) = E_b[\rho_{t:T-1} G_t | S_t = s] \tag{2.14}$$

Now, given trajectory samples from behavior policy b, it's quite straightforward to estimate state-values for target policy.

$$\hat{v}_\pi(s) = \frac{1}{n}\sum_{i=1}^{n}\rho_{t:T-1}G_t \tag{2.15}$$

This estimator is unbiased, but its variance is possibly unbounded. It turns out that there is another way to construct an estimator, which is asymptotically unbiased and possesses much smaller variance. This approach is known as Weighted Importance Sampling [20].

$$\hat{v}_\pi(s) = \frac{\sum_{i=1}^{n}\rho_{t:T-1}G_t}{\sum_{i=1}^{n}\rho_{t:T-1}} \tag{2.16}$$

2.5 Temporal-Difference learning (TD)

TD [26] is one of the central ideas of RL, which is a combination of MC and DP methods. It does not require a model of the environment and can be learned directly from raw experience. MC methods need the whole trajectory to update the estimate of state-value. This makes them hard for tackling continuing tasks. On the other hand, TD methods update estimates based on current reward and next-state state-value estimate - they bootstrap.

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot (R_{t+1} + \gamma \cdot V_(S_{t+1}) - V(S_t)) \tag{2.17}$$

This TD method is called TD(0) and is particularly useful for online learning. As discussed earlier, the advantage of TD over MC is that it is more suitable for continuing tasks. Even if a task is episodic, if the episode length is very long, it's often impractical to use MC methods. Also, TD methods tend to converge faster than MC methods.

A simple GPI-style method, which uses TD(0) for action-values estimation, is called Sarsa. It solves the problem of exploration by using ϵ-soft policies and thus is suboptimal.

Chapter 3

Related Works

The first approach for building dynamic pricing engines was rule-based pricing. Its main idea is to carefully design a set of "if-then" rules to dynamically adjust prices according to different market conditions. Despite its simplicity, it is the most popular and widely used pricing strategy. Competition-based pricing is an example of rule-based pricing. This type of system decides what prices to set based on competitors' prices. It may be implemented in the following way - always set the price to be ten percent lower than the direct competitor's price. After implementing competition-based pricing [5] increases revenue by eleven percent. [4] defines a variance-based pricing rule which guarantees an increase in profit upon fixed per-unit pricing strategy. Another type of rule-based pricing is value-based pricing and cost-based pricing [6]. Despite its simplicity, this paradigm of dynamic pricing possesses a set of drawbacks that make it limited and unscalable. The first drawback is due to the complexity of markets. There are a lot of factors that influence the demand at a specific timestep. These factors may be, company's previous prices, changes in competitors' prices, the time range, advertising spend, inventory level, and other exogenous factors. Having all those input features, manual creation of rule-based formulas for "if-then" logic is an extremely time-consuming process. Those rules should be properly checked, which also requires time and resources. The situation becomes even worse if we consider a few hundred products or services which should be integrated into a dynamic pricing engine. Another drawback of this approach is that it couldn't benefit from new data. The market environment is not stationary in the sense that its properties may change with time. That is why it may be extremely beneficial to integrate new data into the dynamic pricing engine.

Instead of manually defining formulas and rules for "if-then" logic, one could exploit the ML machinery to define a pricing strategy. By using ML models, dynamic pricing engines could handle much more factors, which leads to significantly better performance. Also, ML models could be automatically re-trained or fine-tuned on newly available data without any interventions to the system. While dynamic pricing using ML has many advantages, it also requires sales data as input from the company. Sales data contains prices along with respective sales and, possibly, other features that could be useful in forecasting sales (like inventory, product seasonality, or sales rank). The common way to define a pricing strategy is to build a demand forecasting model and choose a price that leads to the highest income. For example, [3] uses fuzzy rule-based demand forecasting, [15] uses arrival rate estimation to enhance demand forecasting for a real-time dynamic pricing system.

RL methods could be used to solve the problem of greedy decision-making. It considers associativity between the successive pricing stages, thus being non-greedy with respect to current timestep income but greedy with respect to cumulative income. For example, [24] uses RL to tackle a dynamic pricing problem in an electronic retail market. They consider both a single seller and two seller market. [16] presents

an end-to-end RL approach to solve a dynamic pricing problem in e-commerce. They were working with Alibaba marketplace.

Chapter 4

Methodology

4.1 Demand forecasting

The first step in building either demand forecasting based or RL based dynamic pricing engines is to "understand" how the current price affects the sales. This can be achieved by building a demand forecasting model. The model takes as an input a price along with other independent variables and outputs expected sales under such an input. The type of demand forecasting model strongly depends on the dataset and the relationship between dependent and independent variables. As was mentioned earlier, sales data from the Amazon marketplace are used in the experiments. It contains multiple time series. After the data understanding and data preparation steps, we end up with three different daily time series. The first one is a sales time series, which is a target we want to forecast. Inventory and advertising spend are the other two time series. These are exogenous factors, which could be integrated into a forecasting model in order to improve its performance.

Prices time series data was taken from Keepa service. This service allows getting information about a specific product or the whole category of products. For example, it allows getting prices and sales-ranks data for a product by its ASIN. Sales-rank indicator allows comparing different products in terms of how much they are bought. Amazon changes this indicator every hour. Sales-rank may be used as an input to a demand forecasting model too. Keepa also allows getting best-sellers for a specific category. This is a list of ASINs sorted based on the sales-rank indicator.

4.1.1 EDA

In order to capture the proper dependence between price and sales variables, there should be enough variability in the prices. It means that different prices must be present in the dataset, and there should be high enough occurrences of each of them. As seen from the Figure 4.1, prices of a target product are changed rarely and do not possess enough variability. It means that interpolating price data to daily granularity and using it for demand forecasting may be impractical.

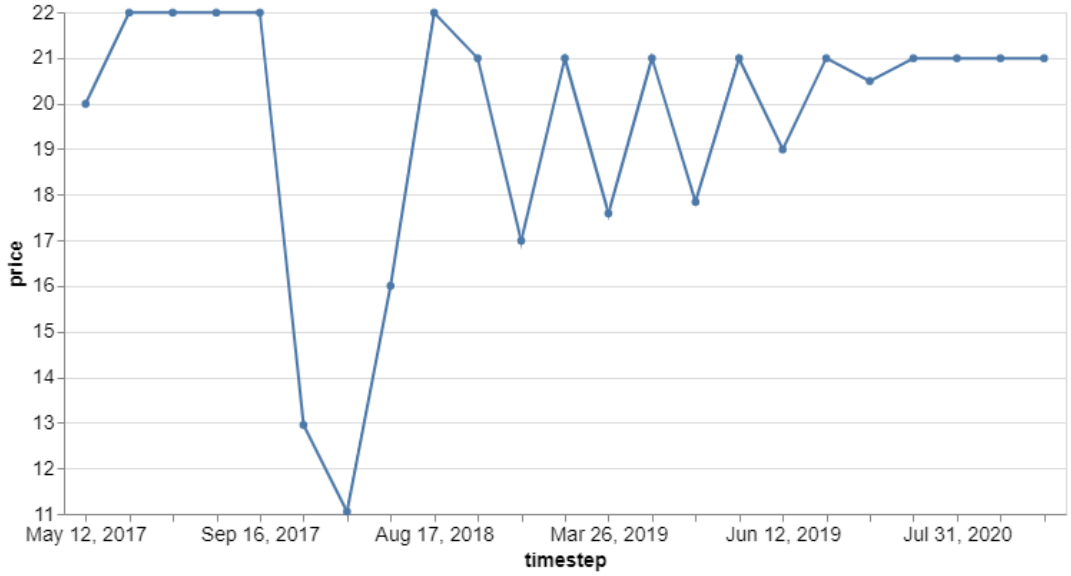


FIGURE 4.1: Target product prices.

To overcome this issue and enrich the price data, competitor's data could be used. To get a competitor list, one could use products that are close enough to the target product in the best-sellers list. As was mentioned earlier, the best-sellers list for a specific category could be taken from a Keepa service. Our target product significantly differs from typical products in its category. That is why, in order to find competitors, we manually filter all products from the best-sellers list and choose similar products. Having a target product and competitors list, one could examine how price affects a sales-rank. Both of these indicators are taken from Keepa. Learning this dependence is a simpler task than learning price-sales dependence because prices of different products are used. This increases the variability of prices and leads to better generalization properties. Knowing how to estimate sales-rank given a price, sales-rank, without a price, could be used as an input to the demand forecasting model. After matching the sales timesteps with sales-rank timesteps, we end up with four time series. A regression model that estimates the sales-rank given price is introduced later in this section.

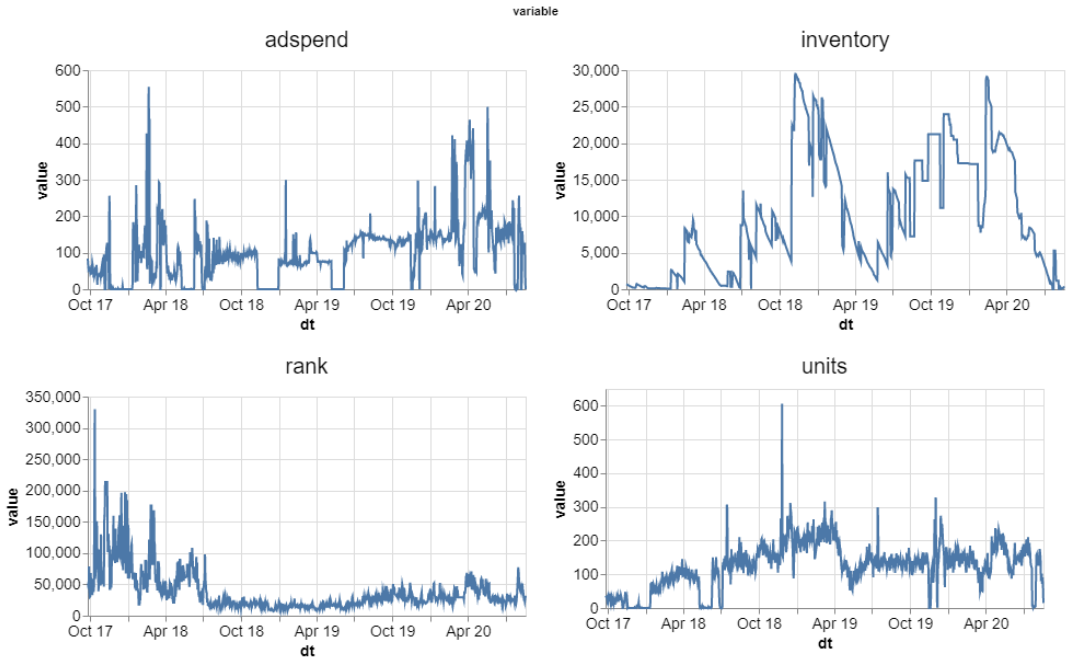
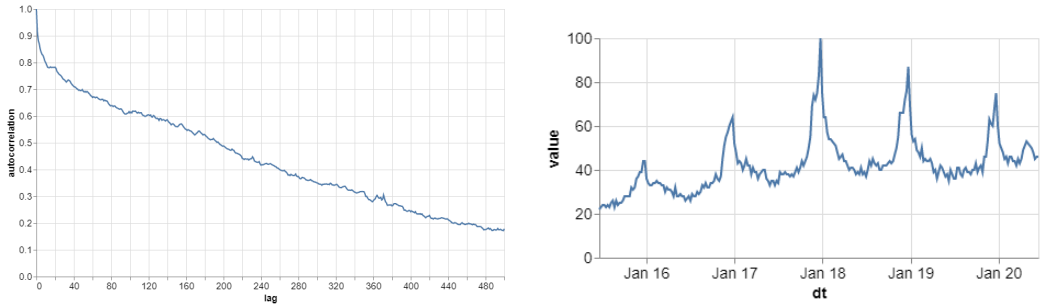


FIGURE 4.2: Sales time series along with exogenous factors.

As we can see from the Figure 4.2, there is no clear trend in the sales. Sales increase, which started in October 2017 and ended in July 2019, could be explained as an increase of inventory and decrease of sales-rank at the respective time range. Also, there is no clear seasonal component seen in the chart. Despite this fact, Google Trends of the target product name shows the seasonality with a one-year period. To verify the hypothesis of one-year seasonality, an auto-correlation function is analyzed. ACF shows the correlation of the signal with the shifted version of itself. As seen from the chart, there is no local maximum around 365 lag. So, we could conclude that there is no seasonal component in the sales data.



(A) Sales ACF (B) Google Trends

FIGURE 4.3: Seasonality hypothesis verification.

In order to evaluate the dependence of sales and other exogenous factors, Pearson correlation and mutual information scores were used. Pearson correlation measures the level of linear dependence between two random variables. It takes values from -1 to 1. This score could be estimated from samples using the following formula

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x}) \cdot (y_i - \bar{(y)})}{\sqrt{\sum_{i=1}^{n}(x - \bar{x})^2} \cdot \sqrt{\sum_{i=1}^{n}(y - \bar{y})^2}}$$

Mutual information score measures the level of nonlinear dependence between two random variables. It takes zero value if and only if two random variables are independent. Higher values correspond to stronger dependence. For continuous random variables, it is calculated as

$$I(X,Y) = \int_X \int_Y p_{(X,Y)}(x,y) \cdot log(\frac{p_{(X,Y)}(x,y)}{p_X(x)p_Y(y)})dydx$$

Estimating mutual information from samples is more complicated than in the Pearson correlation case. It is based on nonparametric methods which use entropy estimation [13]. As seen from the Figure 4.4, there is a dependence between sales and exogenous factors. Advertising spend and inventory are positively correlated with sales, and sales-rank is negatively correlated. All those results yield a clear, intuitive sense.

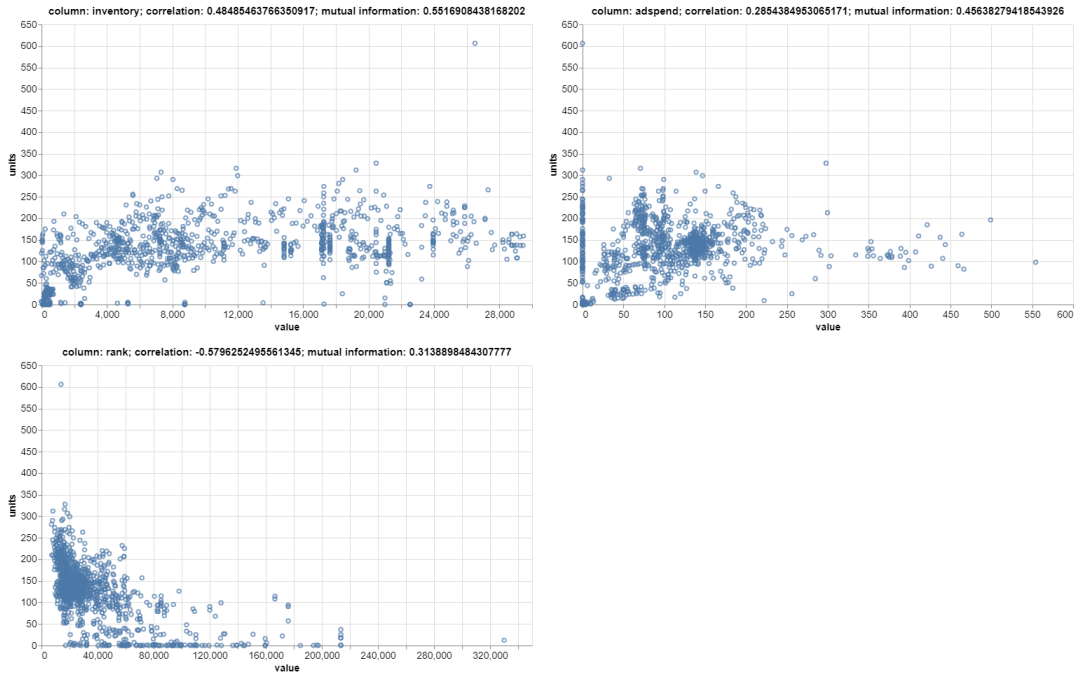


FIGURE 4.4: Dependence of sales and exogenous factors.

Not only current timestep factors may influence sales, but also values from previous timesteps. To define an upper bound on the lags number, random forest feature importance scores and partial autocorrelation function are used. Random forest regressor assigns feature importances based on the mean-squared-error reduction. PACF originally works with a single time series but could be generalized to handle two time series. Roughly speaking, PACF at i's lag shows how "useful" this lag is in case all previous lags are used. To do this, it firstly fits a linear regression model using first $i-1$ lags as features. Then PACF at i's lag is a Pearson correlation between this lag and residuals. Based on obtained results (see Figures 4.5 and 4.6), we decided to use inventory and advertising spend only at zero lag. The upper bound for the number of sales lags is three and for sales-rank lags is fifteen.

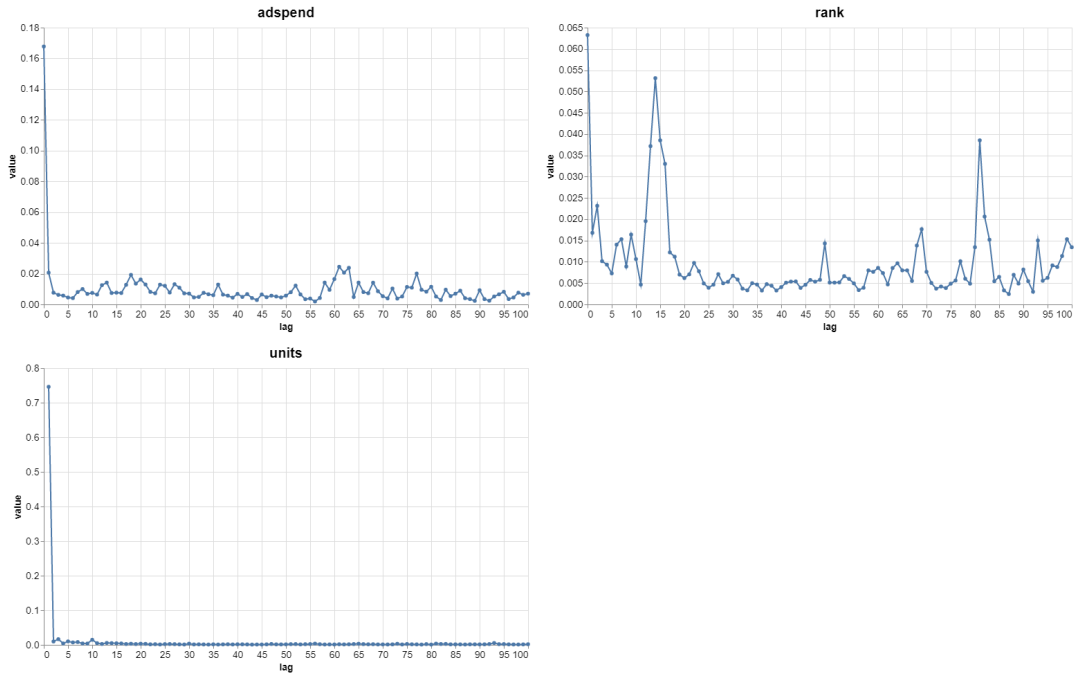


FIGURE 4.5: Random Forest feature importances.

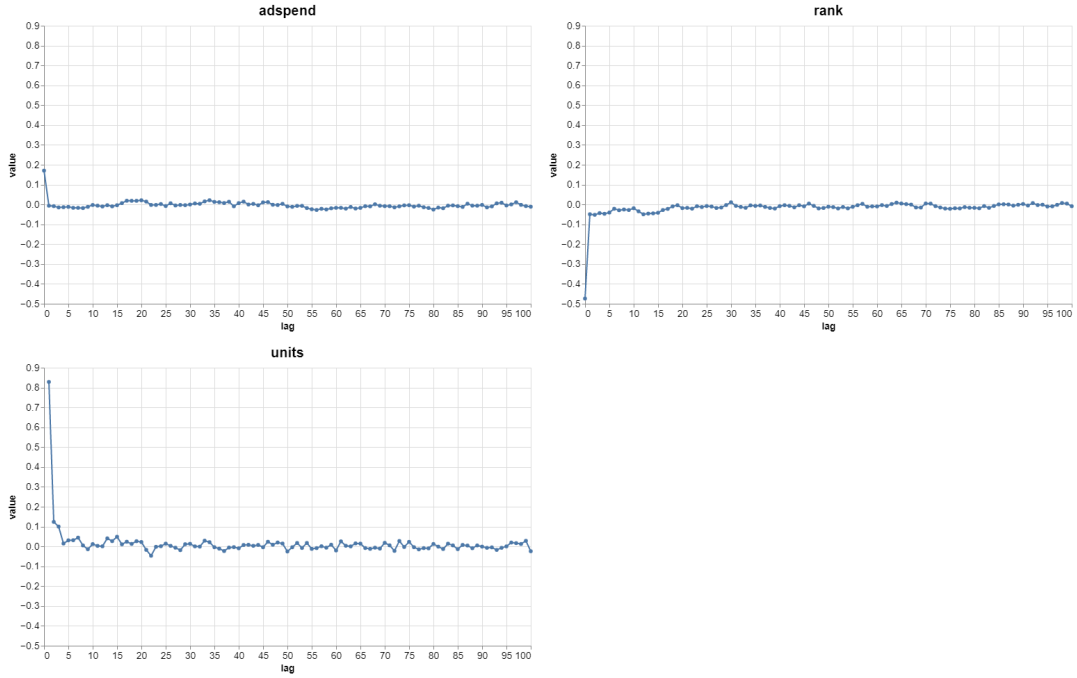


FIGURE 4.6: PACF of sales and exogenous factors.

4.1.2 Models training and validation

Since we use exogenous factors to forecast sales and there are no trend and seasonal components, the usage of ARMA, ARIMA, or SARIMA models is impractical. That is why the ML approach is used to forecast sales. We have tried different ML

estimators, validating different numbers of lags for each time series and different hyperparameters for each ML model.

The standard in ML, K-fold cross-validation is not applicable to time series data. The reason is that one has to preserve the sequential order in the samples. The forecasting model should be trained on the continuous part of the data. The straightforward modifications to the K-fold could be done so that it could properly work with time series data. The basic idea is to define different, mutually exclusive test sets, which are continuous in time. For each test set, a training set is created. The training set is formed by grouping all samples which occurred earlier in time than test-set samples. Finally, different copies of the same model are trained on different training sets, evaluated on respective test sets, and the average metric over all test sets is returned. This validation technique is also called a Walk-Forward validation. The overall process is visualized in the Figure 4.7.

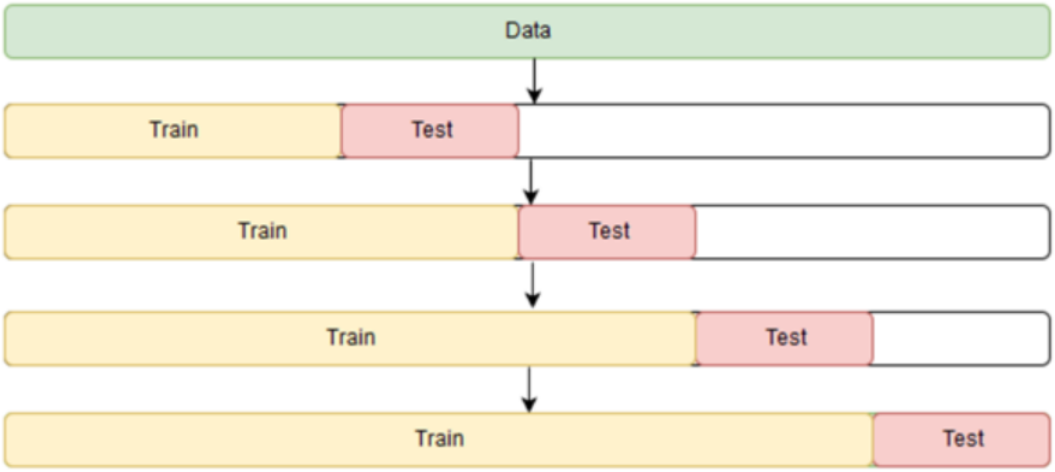


FIGURE 4.7: Time series models validation. The image was taken from [30].

The sales time series has 1058 timesteps. The last 90 timesteps were used for the test set. The test set is used to get an unbiased performance score. Other values were used for models training and validation. We used three folds with 90 samples per fold. The target performance score, which was used to choose the best hyperparameters for each model and the best overall model, is WMAPE. A standard MAPE score is inapplicable because it contains a division by the number of current sales, which can be zero. WMAPE is calculated with the following formula

$$WMAPE = \frac{\sum_{t=1}^{n} |A_t - F_t|}{\sum_{t=1}^{n} |A_t|}$$

Additionally, we calculate the RMSE score. RMSE score has the following formula

$$RMSE = \sqrt{\frac{\sum_{t=1}^{n} (F_t - A_t)^2}{n}}$$

In our experiments, we tried different regression models, such as linear regression, ridge regression, Poisson regression (part of GLM which assumes a dependent variable has Poisson distribution and uses logarithmic link function), support vector machines, random forest, and gradient boosting. For each model, different numbers

Model name	WMAPE	RMSE	Sales max lag	Sales-rank max lag
Linear regression	0.165	31.5	2	5
Ridge regression	0.165	31.5	2	5
Poisson regression	0.197	37.3	3	5
Support Vector machines	0.15	29.7	2	4
Random Forest	0.142	28.2	2	8
Gradient boosting	0.143	28.4	2	8

TABLE 4.1: Validation results of demand forecasting models.

Hyperparameter	max_depth	max_features	min_samples_split	n_estimators
Value	5	0.6	6	200

TABLE 4.2: Random forest hyperparameters.

of lags and different model-specific hyperparameters were used. Validation results are presented in the table 4.1.

As we can see from the Table 4.1, the random forest model has the lowest WMAPE and RMSE scores. This model is used for demand forecasting as the best candidate. Optimal parameters can be seen in the Table 4.2. Hyperparameter "max_features" controls the number of features that are used to build each separate decision tree. In our case, the previous sales are highly correlated with current sales. It can be seen from the PACF chart in the EDA section. If all decision trees use this feature, they will have similar splits and, thus, their predictions will be correlated. This could lead to overfitting. As we can see, random forest "figures out" that it is beneficial to use only 60% of features for each decision tree.

WMAPE on the test set is equal to 0.16, RMSE equals 26.6. WMAPE slightly increases on the test set, but RMSE becomes even smaller. This means that the random forest did not overfit the training set and possesses acceptable generalization properties. Predictions on the test set can be seen in the Figure 4.8. Finally, the random forest is re-trained on the whole dataset using optimal hyperparameters and the number of lags.

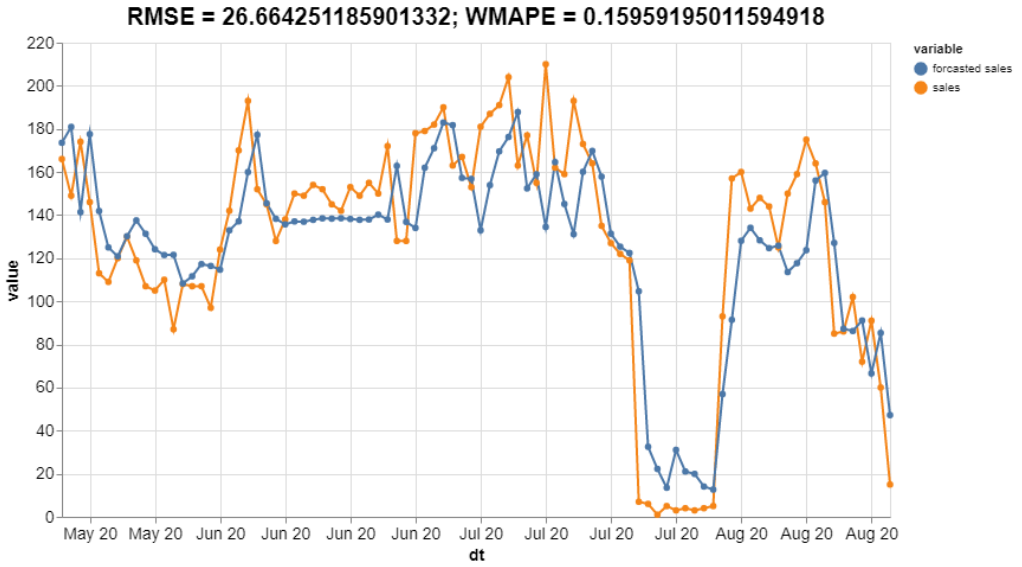


FIGURE 4.8: Predictions on the test set.

4.1.3 Sales-rank prediction

As was mentioned earlier, sales-rank, instead of price, is used as an input feature to the demand forecasting model. In order to understand how prices affect sales, one has to build a predictive model that inputs a price and outputs the expected sales-rank. Price-rank pairs are generated from both the target product and competitors' products. After discretization of prices and averaging sales-ranks for each possible price, we end up with the following set of points.

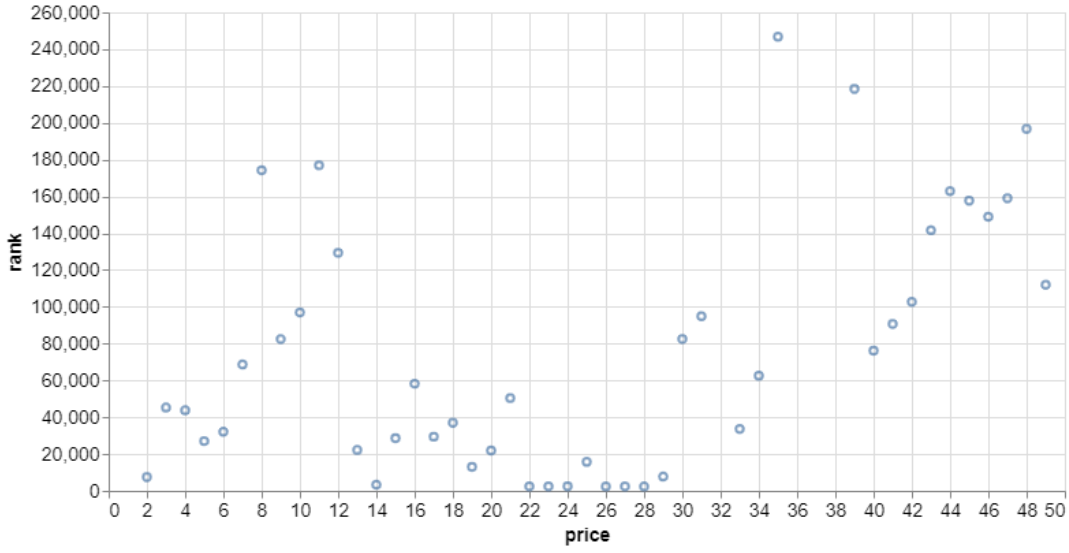


FIGURE 4.9: Price-Rank dependence.

These points can be fitted with the polynomial functions. Polynomial parameters are estimated using the ordinary least squares method. Using different polynomial degrees yields the following curves

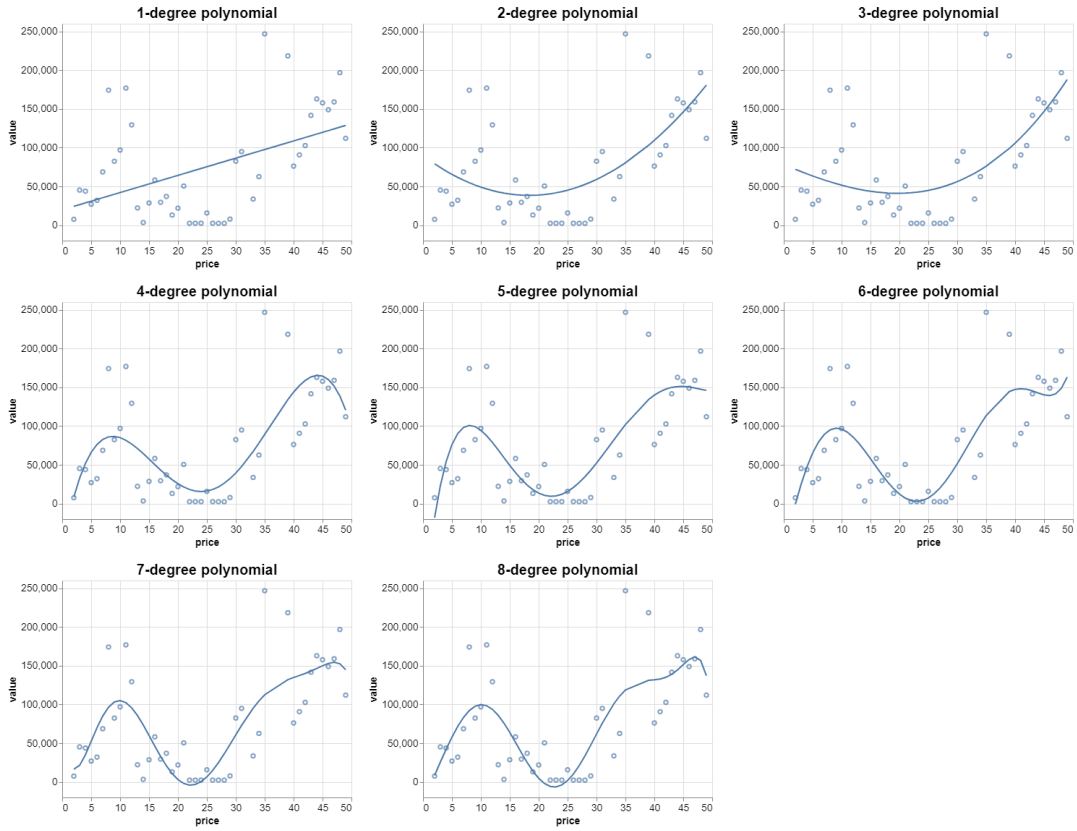


FIGURE 4.10: Different polynomials fit.

A six-degree polynomial is used as the final model. It adequately approximates the data and does not decrease too much at the end of the interval.

4.2 Demand forecasting based pricing strategy

By integrating dynamic pricing engines, companies are following their financial goals. For simplicity, we assume that the economic performance indicator, which the company is willing to maximize, is the revenue. Having a demand forecasting model, the company could estimate how different prices affect the revenue at the current timestep. For example, if one possesses all input features to the forecasting model except the current price, different prices could be plugged in, and respective revenues (or any other performance metric) could be calculated. Finally, the price that leads to the highest revenue is chosen. In our case, if possessing previous sales, previous sales-ranks, current inventory level, and current advertising spend, one could pick up different prices, estimate sales-rank using a sales-rank prediction model, and feed all those input features to the demand forecasting model. Having estimated sales number, respective revenues could be calculated. This pricing strategy is also called a greedy strategy, and the respective price is called greedy because it maximizes the revenue at the current timestep and does not account for future timesteps.

The problem of finding a greedy price can be formulated as a one-variable optimization problem over a continuous interval. Interval bounds are defined by a

minimum and maximum price. Since a demand forecasting model could be non-parametric and use non-differentiable operations, gradient-based optimization techniques, such as gradient ascent, Newton method, or trust region optimization, are not applicable. In our case, a random forest regressor is used to forecast sales. This model is non-parametric, and its output couldn't be differentiated with respect to its input. That's why a typical solution is to discretize a continuous interval of possible prices. Then for each discretized value, sales and revenue estimates are calculated. The price which maximizes the revenue is used. An additional advantage of this method is that it couldn't stack in a local minimum.

The drawback of this approach is that it is greedy. In other words, it considers only revenue in a current timestamp rather than cumulative revenue over a more extended period of time, which is a better choice for maximization. Current prices may influence future market conditions, which may be significantly better for a company. To illustrate this problem, let's consider the Figure 4.11. The agent needs to perform two consecutive actions. By acting greedy with respect to immediate reward, the agent cumulatively earns 18 units. On the other hand, by acting non-greedy at the first timestep, the agent is able to earn 103 units in total. This problem could be tackled by using RL based pricing strategy.

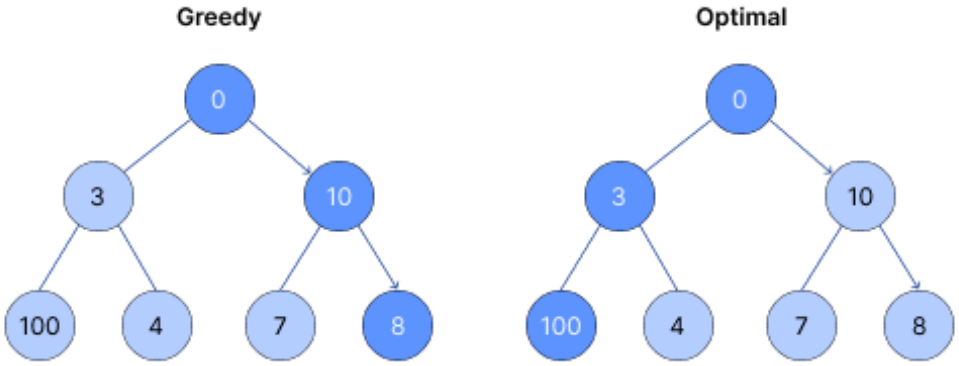


FIGURE 4.11: Greedy vs Optimal strategies.

4.3 RL based pricing strategy

RL methods are the alternative approach that solves the problem of greedy decision-making. These methods consider associativity between the successive pricing stages, thus acting non-greedy with respect to current timestamp revenue but greedy with respect to cumulative revenue. For example, it may be useful to put a lower price in the current timestamp, decreasing the immediate revenue but attracting more customers. This may lead to a significantly better market state in the future. Dynamic pricing engines select optimal prices based on market conditions. In terms of MDP, actions are prices, and states are market conditions, except for the current price of the product or service.

Usually, it is incredibly problematic to train RL agents from the interaction with a real market. The agent must get lots of samples from an environment in order to approximate the optimal policy. This may be a very time-consuming process. Another problem is an exploration-exploitation trade-off. While learning an optimal policy, an agent must visit a representable subset of the whole state space, exploring different actions. Consequently, an agent will definitely act sub-optimally while training and could lose lots of money for a company. Basically, time and money limitations do not allow RL agents to be fully trained in the real market environment. That is why the agents are usually pre-trained on a simulated market environment and further fine-tuned while interacting with the real market.

In order to simulate the environment, the state-space and action-space should be defined. After that, the reward signal should be modeled as a stochastic function defined on the cartesian product of state and action spaces. Finally, the transition between states should be modeled in the simulator. Using a demand forecasting model, the expected reward (for example, revenue) can be estimated. As a consequence, input features for the demand forecasting model should be a part of the state representation. Modeling transition between states strongly depends on the task, but it tends to make a few modeling assumptions to be solved. Because of the complexity of the market, it is extremely hard to simulate it accurately. This is the main drawback of RL methods. In the next subsection, there is an explanation of the market simulator.

4.4 Market simulation

The state representation, along with the process of reward calculation and next state transitioning, are described in this section. As was mentioned earlier, the expected reward could be estimated using a demand forecasting model. The problem is that, among input features for a demand forecasting model, there are exogenous factors, which change in time. These changes may depend on other factors which should be separately modeled and integrated into the simulator. Roughly speaking, the state contains two separate sets of features. The first set consists of all input features for the demand forecasting model, except the current sales-rank. The second set consists of features that are used to change exogenous factors from the first set. The actions are all possible prices.

4.4.1 Reward signal

An immediate reward at each timestep is a performance metric, which the company is willing to maximize. In our case, we assume that it is revenue. An environment, having an action, could estimate the sales-rank. Then by combining the sales-rank with part of the state that corresponds to demand forecasting, the expected sales number could be estimated using a demand forecasting model. The estimated sales-rank goes to the next state representation as the previous sales-rank, and all other sales-ranks are shifted by one, ignoring the earliest sales-rank.

Demand forecasting model outputs expected sales. Obviously, the number of sales is influenced by an enormous amount of factors, which we don't account for. That is why in practice, we could receive different sales even though the values of considered features are the same. To handle this problem, the errors of the demand forecasting model on the test set are examined. The idea is to fit those errors with

parametric distribution, and each timestep sample an error term from this distribution, adding it to the estimate of the expected sales. As seen from the chart, the distribution of errors has a bell-shaped form and could be approximated with the Gaussian distribution. In order to verify the hypothesis of normality, the Shapiro-Wilk test is used. 1% significance level is used for all conducted tests in this thesis. After calculating the p-value, the null hypothesis was not rejected, and the errors were modeled with Gaussian distribution. After sampling the sales, revenue can be calculated and returned to the agent. Sampled sales go to the next state representation as the previous sales. As in the sales-rank case, the earliest sales are not included in the next state.

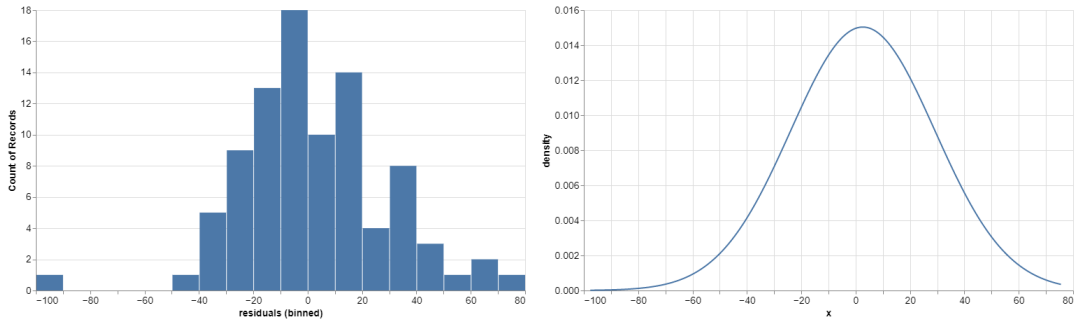


FIGURE 4.12: (Left) Distribution of sales residuals. (Right) pdf of estimated Gaussian distribution.

4.4.2 Advertisement spend change

Since the current advertising spend is used as a feature to forecast the demand, it should be somehow changed in the environment. It was decided to estimate the next advertising spend based on previous advertising spends and previous sales. In order to choose the number of lags, PACF is analyzed. As seen from the Figure 4.13, the first lags have the highest PACF, and all other lags are small enough to ignore them.

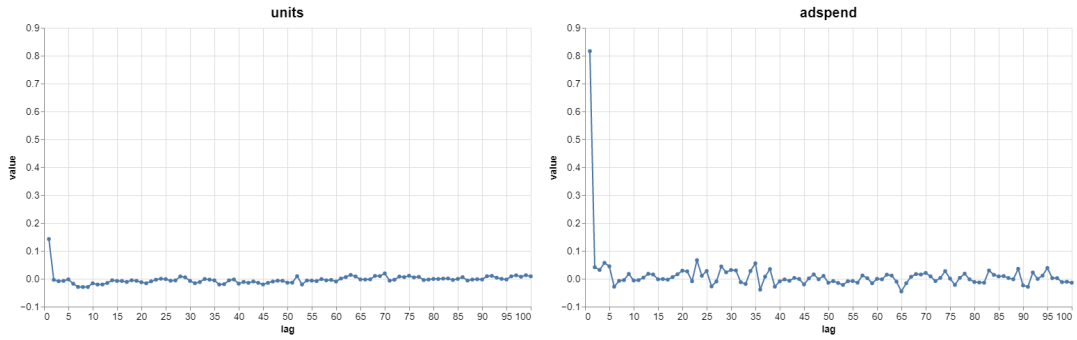


FIGURE 4.13: PACF of advertising spend with itself and with sales.

Advertising spend has a clear linear relationship with its first lag. It can be seen in the Figure 4.14.

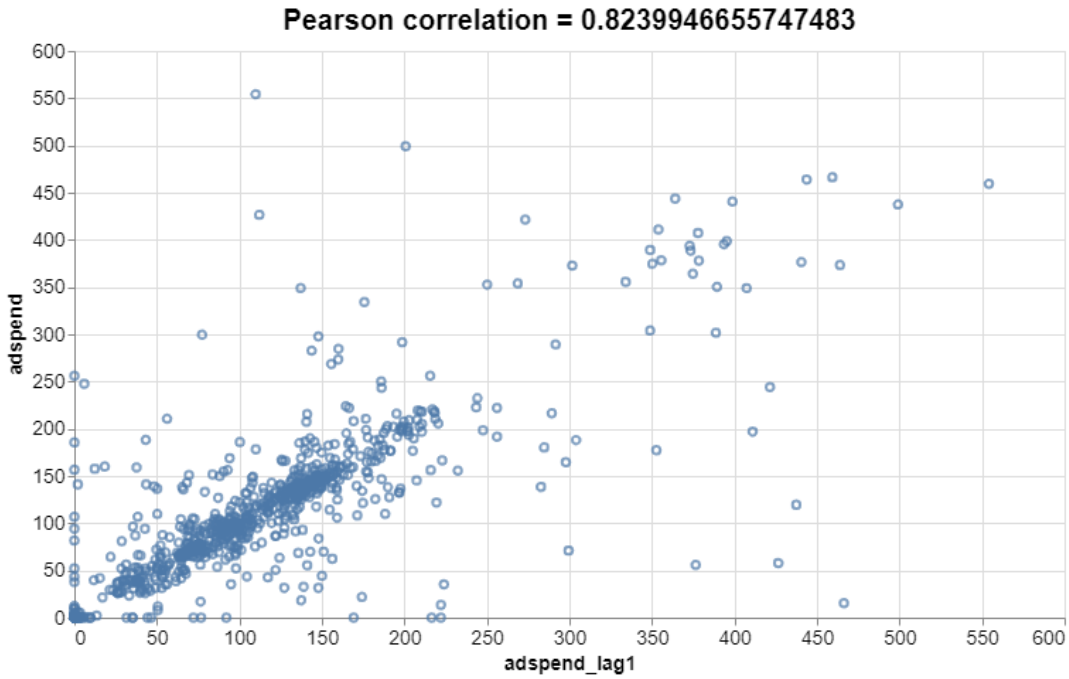


FIGURE 4.14: Dependence of advertising spend with its first lag.

After fitting a linear regression model using lag-1 advertising spend as a single independent variable, the correlation of lag-1 sales with residuals is calculated. As seen from the Figure 4.15, after using lag-1 advertising spend, lag-1 sales have a very small correlation with the residuals. It can be a sign of conditional independence of current timestep advertising spend and previous timestep sales given previous advertising spend. That is why the final model for predicting the expected advertising spend is a linear model that takes as an input only previous advertising spend.

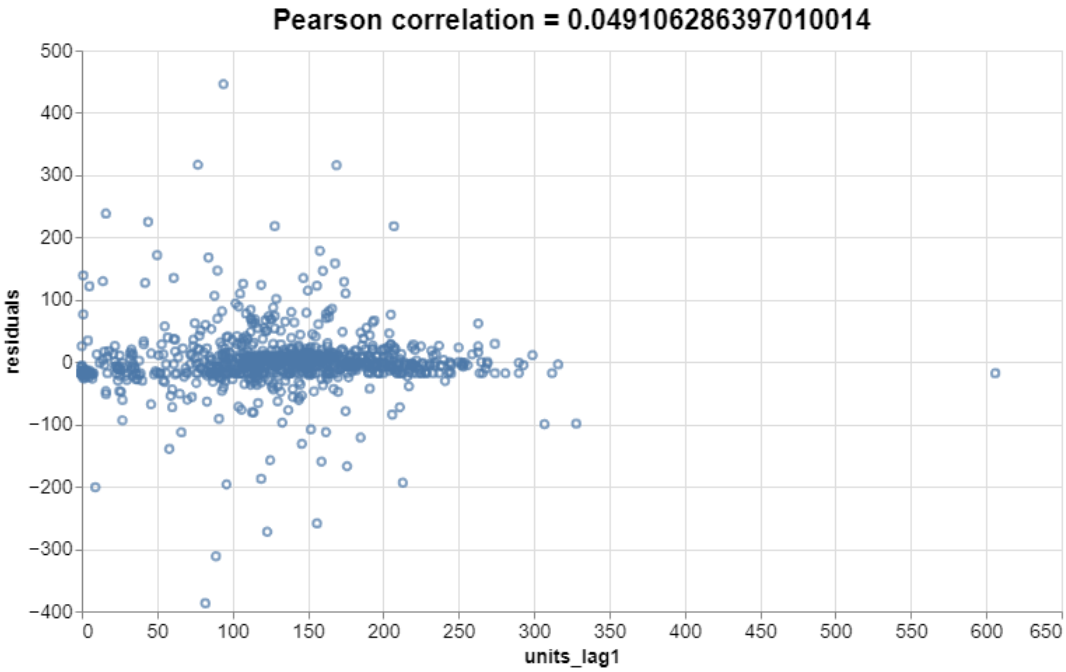


FIGURE 4.15: Dependence of sales and residuals.

Following the same arguments as in the sales forecasting case, the prediction errors are fitted with a parametric distribution. After applying the Shapiro-Wilk test, the null hypothesis of normality is rejected. That is why the Laplace distribution (also called a double exponential distribution) was used to model the residuals data. The environment takes the current advertising spend and estimates the next timestep advertising spend by applying a linear regression model and adding an error term sampled from the Laplace distribution. Estimated advertising spend replaces the current advertising spend in the next state.

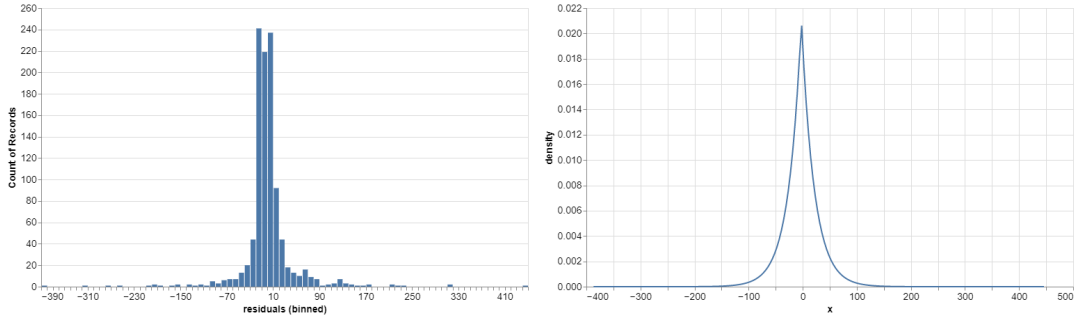


FIGURE 4.16: (Left) Distribution of advertising spend residuals. (Right) pdf of estimated Laplace distribution.

4.4.3 Inventory change

Inventory level indicator should not be forecasted. The company has well-defined policies for producing and delivering its products. These processes can be simulated in the environment. In our case, for the sake of simplicity, we assume that the warehouse is supplied with new products at most one time per month. The logic,

implemented in the environment, is the following. At each timestep, the environment looks whether the delivery was executed in the current month. Then if new products were not supplied yet, it performs a supply with some probability p. Supply probability is chosen in a way that the mean number of supplies per month is equal to the real value, which is estimated from the historical data. In order Markov property is satisfied, two additional variables are added to the state representation. These variables are day number and boolean variable, which shows whether supply already was in the current month. This boolean variable is reset every new month. The supply amount is sampled from an exponential distribution. Its parameters were estimated from the historical data. The distribution of supply amounts and the pdf of estimated exponential distribution can be seen in the Figure 4.17.

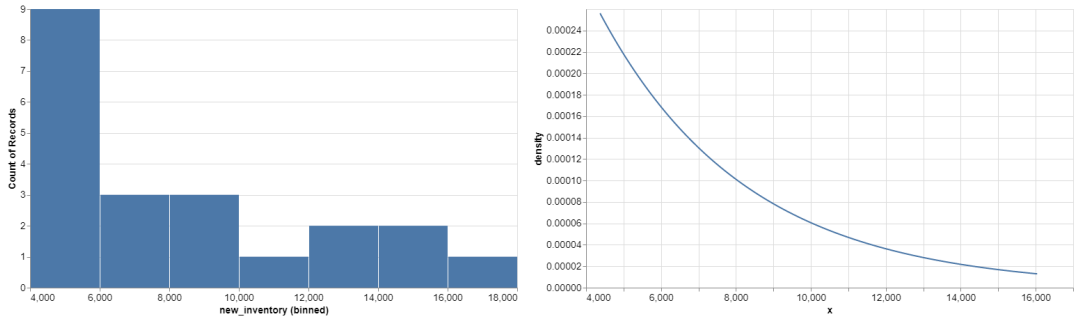


FIGURE 4.17: (Left) Distribution of supply amounts. (Right) pdf of estimated exponential distribution.

4.5 Tabular Q-Learning

Q-learning is an off-policy TD control algorithm. It iteratively finds the optimal action-values.

$$Q_*(s,a) = \max_\pi E_\pi[G_t|S_t = s, A_t = a]$$

As stated earlier, the optimal policy can be easily reproduced from optimal action-values. As was stated earlier, it would be any greedy policy with respect to optimal action values. The following formula is used to update estimates

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot (R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a) - Q(S_t, A_t)) \tag{4.1}$$

The estimates converge to the optimal action values, independent of the policy used. Usually, the epsilon-greedy policy with respect to the current estimates is used. This update formula can also be treated as an iterative way of solving Bellman optimality equations.

This algorithm assumes a discrete state-space and discrete action-space. Accordingly, before running this algorithm, one should discretize continuous variables into bins. The discretization rate is a hyper-parameter of the algorithm and should be tuned while training. The name "tabular" means that estimates of action-values are stored in one huge table. Memory usage and training time grow exponentially with the increase of features in the state representation, making it computationally intractable for complex environments (for example, Atari games). In order to solve

Algorithm 1 Tabular Q-Learning

Discretize continuous variables and initialize q-table $Q(s, a)$.
Initialize the learning rate α.
for $episode = 1, 2, \ldots$ **do**
    Obsereve the initial state S_0.
    for $t = 1, 2, \ldots, T$ **do**
        Perform action A_t using ϵ-greedy policy.
        Observe R_{t+1} and S_{t+1}.
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot (R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$
    end for
end for

this problem, function approximation [21] could be used to represent optimal action values. This allows a more compact representation that does not depend on state-space size.

Both Sarsa and Q-learning involve maximization in the construction of their target policies. A maximum over the estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias. This problem is called a Maximization Bias. To tackle this problem, two separate estimates are used. One of them is used for estimating greedy action, and another for estimating its action-value. This idea is called Double Q-learning [7]. At each timestep, only one random table is updated.

4.6 Deep Q-Network

DQN [9] algorithm is an action-values method. It is based on the Q-learning idea of finding optimal action-values by using off-policy TD control methods. The main difference to the tabular approach is that DQN uses a parameterized function to approximate optimal action-values. More specifically, DQN uses ANNs as approximators. Based on state representation, both CNNs and RNNs can be used. For example, in case an agent learns to play video games, CNNs could be used to extract features and predict optimal action-values from raw images. In case a few consecutive frames are needed to represent a state, so that a Markov property is held, DQN could use CNNs for feature extraction and RNNs (or any modification like LSTM [8] or GRU [1]) to process image representations in a sequential manner and output an estimate of optimal action-value.

As was mentioned earlier, the problem with the tabular approach is that its time and memory complexity exponentially increases with the increase of state-space dimensions. In case we are using function approximation to represent optimal-action values, we significantly reduce our memory consumptions. Now, one has to store a single approximator in RAM. Its size may also depend on the number of state representation dimensions, but it is not an exponential relation.

To obtain reliable estimates of optimal action-values for each state-action pair, the agent needs to discover all state-space and action-space. If two states are similar in terms of state representation, tabular Q-learning should learn its action-values independently without any information sharing. This is due to the fact that each action-value is stored as a separate cell in a table. A benefit of using function approximation that allows reducing time complexity and reducing the number of interactions with an environment is that it could learn the optimal action-value of a

state without actually exploring this state. The reason is the following since we use some parameterized function to obtain optimal action-value from a state-action pair, in theory, an agent could explore a representable subset of state-space and generalize to unseen states because of shared parameters. Also, it allows handling continuous state-spaces out-of-the-box.

Before using ANNs as approximators, more simple models that allow stochastic optimization were used. For example, [12] uses a linear model on top of coordinates in a Fourier basis. RBF and polynomial basis functions were also used. In the case of complex environments, these models don't possess enough capacity to approximate optimal action-values. For example, it is impractical to input raw images to these types of models. To tackle this problem, one should create a more compact state representation by doing feature engineering.

There are two possible options how to construct ANN architecture:

1. The first option is that ANN takes as an input a state and action and outputs an estimated optimal action-value.
2. The second option is that ANN takes only a state as an input and outputs estimated optimal action-values for each action.

The second approach assumes that there is the same number of actions for each state. If this assumption holds, it is a preferable architecture since it allows estimating a greedy action in a single forward pass. This could significantly speed up learning. The first option is the basis for the DDPG [14] algorithm and could be used in the case of continuous action-spaces.

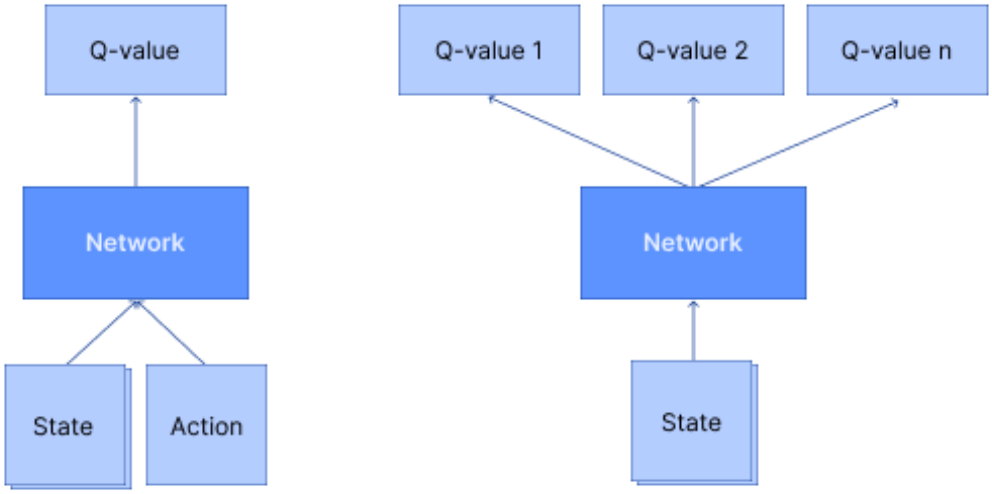


FIGURE 4.18: ANN architectures.

What we did with Q-learning is to iteratively update action-value estimates with the following formula (4.1). Basically, at each timestep t we make estimate of (S_t, A_t) pair a little bit closer to the target y_t.

$$y_t = R_{t+1} + \max_a Q(S_{t+1}, a) \tag{4.2}$$

The problem with DQN and any other function approximation approach is that we could change our estimates only by changing the parameters of the model. The

straightforward solution is at each timestep to perform a stochastic gradient descent step with respect to model parameters based on four-pair $S_t, A_t, R_{t+1}, S_{t+1}$. To do this, we will define a loss function at iteration i

$$L_i(\theta_i) = E_{s,a\sim p(.)}[(y_i - Q(s,a;\theta_i))^2] \tag{4.3}$$

where

$$y_i = E_{s'\sim\varepsilon}[r + \gamma \cdot \max_{a'}(Q(s',a';\theta_{i-1}))|s,a]$$

The distribution p is obtained by acting ϵ-greedy with respect to current action-value estimates.

Then the gradient of the objective has the following form

$$\nabla_{\theta_i} L(\theta_i) = E_{a\sim p(.);s,s'\sim\varepsilon}[(r + \gamma \cdot \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)] \tag{4.4}$$

A single-sample approximation is used. This leads to the usual Q-learning update formula. This simple extension of the Q-learning algorithm possesses two drawbacks which make it very hard to train ANN. These drawbacks are the following:

1. Using stochastic gradient descent assumes consecutive samples to be independent and drawn from the same distribution. Obviously, this assumption is violated.

2. ANN could perform worse on states that did not occur for a long time.

Both of these problems are solved by using an experience replay mechanism. The idea is the following: at each timestep, save a four-pair $(S_t, A_t, R_{t+1}, S_{t+1})$ to the buffer and sample from this buffer a batch of four-pair samples. These four-pair samples are used for mini-batch gradient descent update of parameters. This algorithm allows using a single sample multiple times and thus is more sample efficient.

Another possible problem is connected with autocorrelation. By changing the parameters of the model, we change the targets yi for future timesteps. This could harden the learning process. The solution is to have another network - target network, which is used for calculating the target. After a predefined number of interactions with an environment, parameters from our training network are copied to the target network so that they are up to date.

Since DQN is a modification of Q-learning, it also has a problem of Maximization Bias. In Q-learning, we tackled this problem by using two separate tables of estimates. Since we integrate the target network into our algorithm, we should not create another network to tackle the maximization bias problem. Now, we could estimate optimal action with target network and optimal action-value with prediction network.

4.7 Policy Gradients

The policy gradients algorithm is based on an entirely different idea of learning an optimal policy. Instead of estimating optimal action-values, policy gradients directly parametrize and optimize a policy. ANNs could be used to parametrize a policy.

$$\pi(a|s,\theta) = P(A_{t+1} = a|S_t = s, \theta_t = \theta) \tag{4.5}$$

Algorithm 2 DQN

1: Discretize action-space if needed.
2: Initialize $Q(s, a; \theta)$ with random weights.
3: Initialize an experience replay buffer and learning rate α.
4: **for** $episode = 1, 2, \ldots$ **do**
5: Obsereve the initial state S_0.
6: **for** $t = 1, 2, \ldots, T$ **do**
7: Perform action A_t using ϵ-greedy policy.
8: Observe R_{t+1} and S_{t+1}.
9: Save four-pair $(S_t, A_t, R_{t+1}, S_{t+1})$ to experience replay.
10: Sample mini-batch from experience replay.
11: Calculate y_t's using formula (4.2).
12: Perform a gradient descent step on $(y_t - Q(S_t, A_t; \theta))^2$
13: **end for**
14: **end for**

This could be beneficial because of a few reasons:

1. An optimal policy itself may be a simpler function than an optimal action-values function.

2. Policy gradients could handle continuous action spaces without a need for discretization.

3. Policy gradients possess better convergence properties because of smooth policy updates (in action-value methods, a small estimate change may lead to a significant change of policy).

The general idea is to parametrize a policy and define some objective we want to maximize, which is a function of policy and thus its parameters. Basically, this function shows how good an input policy is. This objective function is optimized with stochastic gradient ascent until convergence. The gradients of the objective are estimated while the interaction with an environment.

For simplicity of the derivations, we will restrict ourselves to episodic tasks. Also, we will assume that each episode starts in some non-random state s0 (all results are valid if we relax this assumption). Then the objective function is defined as follows:

$$J(\theta) = v_{\pi_\theta}(s_0) \tag{4.6}$$

The problem is that a gradient of the objective depends on the dynamic function p, which, in a general case, is unknown. Policy gradient methods exploit the result of the policy gradient theorem, which allows rewriting the gradient of the objective as an expected value, which could be approximated while acting in the unknown environment. Policy gradient theorem states that

$$\nabla J(\theta) = E_\pi[\sum_a q_\pi(S_t, a) \nabla \pi(a|S_t, \theta)] \tag{4.7}$$

Then stochastic update of the parameters looks as follows:

$$\theta_{t+1} = \theta_t + \alpha \sum_a \hat{q}(S_t, a; w) \nabla \pi(a|S_t, \theta) \tag{4.8}$$

This update requires having estimates of all action-values and uses all actions to perform an update. This method is called the all-actions method.

Another option is to further simplify the form of the gradient so that we don't require having action-value estimates. REINFORCE [31] algorithm uses the following formula to estimate the gradient of the objective function:

$$\begin{aligned}\nabla J(\theta) &= E_\pi[\sum_a \pi(a|S_t,\theta) q_\pi(S_t,a) \frac{\nabla \pi(a|S_t,\theta)}{\pi(a|S_t,\theta)}] \\ &= E_\pi[q_\pi(S_t,A_t) \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}] \\ &= E_\pi[G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}]\end{aligned} \tag{4.9}$$

In this case update formula takes the following form

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \tag{4.10}$$

REINFORCE algorithm could be used to solve episodic tasks. As in MC methods, this algorithm needs to run the whole episode to calculate a return. This fact leads to high variance of an estimator and could significantly slow down the training process. To reduce the variance of the estimator, the baseline could be used. The idea is to change the estimator so that it remains unbiased but possesses a smaller variance. We could subtract baseline function b, which depends only on the state, from return without affecting a mean value

$$\nabla J(\theta) = E_\pi[\sum_a (q_\pi(S_t,a) - b(S_t)) \nabla \pi(a|S_t,\theta)] \tag{4.11}$$

The estimator remains unbiased since

$$\sum_a b(s) \nabla \pi(a|s,\theta) = b(s) \nabla \sum_a \pi(a|s,\theta) = b(s) \nabla 1 = 0 \tag{4.12}$$

Then update formula takes the following form

$$\theta_{t+1} = \theta_t + \alpha (G_t - b(S_t)) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \tag{4.13}$$

State-value estimates are usually used as a baseline. They are learned simultaneously with optimal policy. The overall algorithm looks as follows:

Algorithm 3 REINFORCE with Baseline

Initialize $\pi(a|s,\theta)$ and $\hat{v}(s,w)$ with random weights.
Initialize learning rates α^θ and α^w.
for *episode* $= 1, 2, \ldots$ **do**
    Generate an episode $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ..., R_T$ using policy π.
    for $t = 1, 2, \ldots, T-1$ **do**
        $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-(t+1)} R_k$
        $w \leftarrow w + \alpha^w (G_t - \hat{v}(S_t,w)) \nabla \hat{v}(S_t,w)$
        $\theta \leftarrow \theta + \alpha^\theta (G_t - \hat{v}(S_t,w)) \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$
    end for
end for

To handle continuing tasks, Actor-Critic methods are used. Its main difference to REINFORCE is that it bootstraps. Since G_t could not be calculated directly, Actor-Critic methods approximate G_t as

$$\hat{G}_t = R_{t+1} + \gamma \cdot \hat{v}(S_{t+1}, w)$$

In this case update formula looks as follows

$$\begin{aligned} \theta_{t+1} &= \theta_t + \alpha(\hat{G}_t - \hat{v}(S_t, w)) \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \\ &= \theta_t + \alpha(R_{t+1} + \gamma \cdot \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)) \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \end{aligned} \tag{4.14}$$

Chapter 5

Experiments

The simulated market environment has a 14-dimensional state-space. The action-space is a continuous interval of possible prices. The minimum and maximum actions are 2 and 49, respectively. These values are taken as a minimum and maximum price over the whole list of competitors. It was decided to use an episodic scenario for experiments. The length of the episode is 30, which corresponds to one month. The starting states are uniformly sampled from the historical data. The reason for using an episodic task is that the simulation becomes less accurate with time due to autoregression. The environment calculates estimates based on previous estimates. Another benefit of episodic setting is that it allows companies to directly control the level of the greediness of their dynamic pricing engines.

We did not use any third-party libraries for training RL agents. The training code was implemented on the solid PyTorch framework. The idea of implementation was that the entire experiment is set up with a single config file. Config file encapsulates all required information, such as hyperparameters of the method, training parameters, and folder with logs. Training logs are saved into the TensorBoard. This allows analyzing conducted experiments and choosing between different agents. The program also monitors the average return over a predefined number of the last episodes. Copy of the model, which maximizes this score, is saved to disk. The interface of agents is unified across different methods. Because of this, the evaluation of different agents could be done with a single code. Method name and path to the saved model are the only things that should be specified in the evaluation config file.

5.1 Agents training

5.1.1 Tabular Q-Learning

The tabular Q-Learning method is of limited use because of memory restrictions. All continuous features were discretized into four intervals. The q-table occupies 2GB. In the case of using five intervals, the q-table occupies over 40 GB and could not be fit into the RAM. Despite the small number of bins, the agents were able to improve their performance, producing quite smooth learning curves. This could be seen from the Figure 5.1.

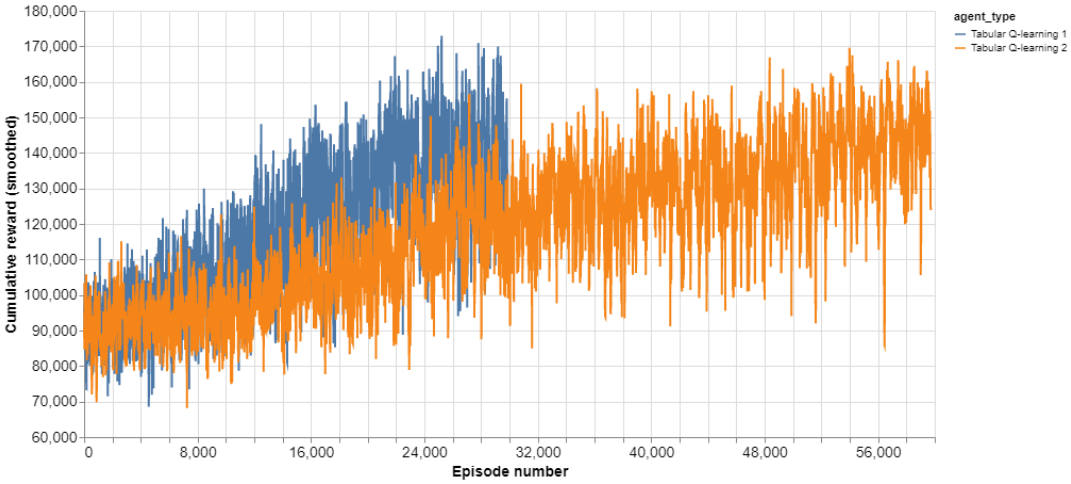


FIGURE 5.1: Learning curves of Tabular Q-Learning agents

5.1.2 DQN

Because of parameter sharing, DQN requires far fewer episodes to be trained. Another advantage is that it does not require discretization of input features. Different numbers of hidden layers were tried. We stop on two hidden layers with 32 neurons on each. Also, different numbers of bins for action-space discretization were tried. We stop on 10 bins since higher numbers do not improve the performance. The crucial hyperparameters, which allowed agents to start training, are the size of the replay buffer and the number of timesteps between copying weights from training to target networks. We end up with the size of replay buffer equal to 250 and copy timesteps equal to 90. As seen from the Figure 5.2, the learning curves have quite big fluctuations. It is due to the fact that a small change of parameters can lead to a significant change of policy.

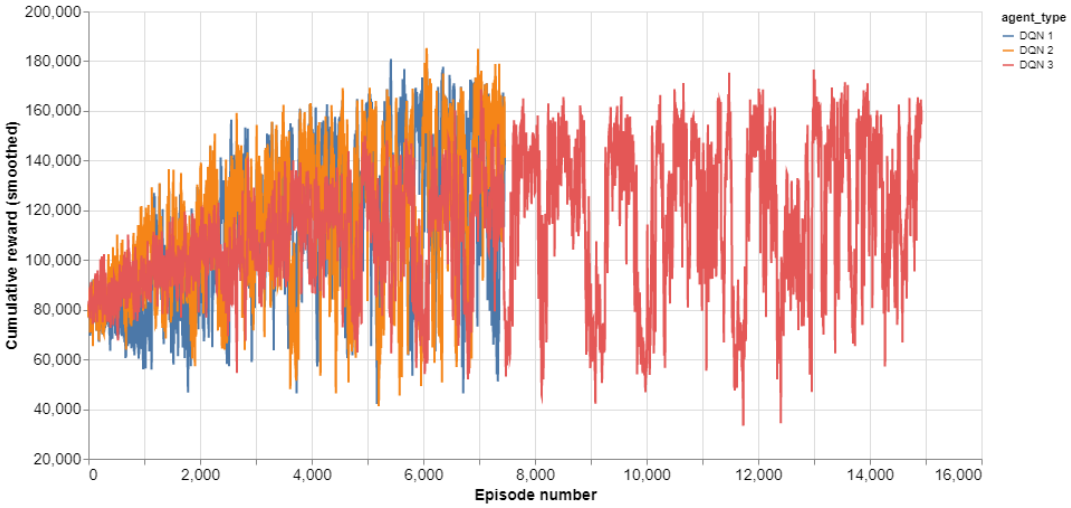


FIGURE 5.2: Learning curves of DQN agents

Agent type	Random	Greedy	Tabular	DQN	Policy Gradient
Mean return	74613.6	117489.2	135129.5	154276.6	149189.9

TABLE 5.1: Mean returns over 300 episodes

5.1.3 Policy Gradient

REINFORCE with baseline algorithm was used for experiments. State-value estimates were used as a baseline function. Two separate fully-connected neural networks approximate the optimal policy and its state-values. They share a set of common hidden layers. Different numbers of common layers and different architecture of both heads were tried. We stop on two common layers with 64 and 32 neurons. Both a state-value head and a policy head have a single layer. As in the DQN, we discretize action-space into 10 intervals. As seen from the Figure 5.3, policy gradient agents produce the smoothest learning curves. It is due to the fact that, unlike in DQN, a small change of parameters leads to a slightly changed policy. On the other hand, policy gradients require more episodes to be trained.

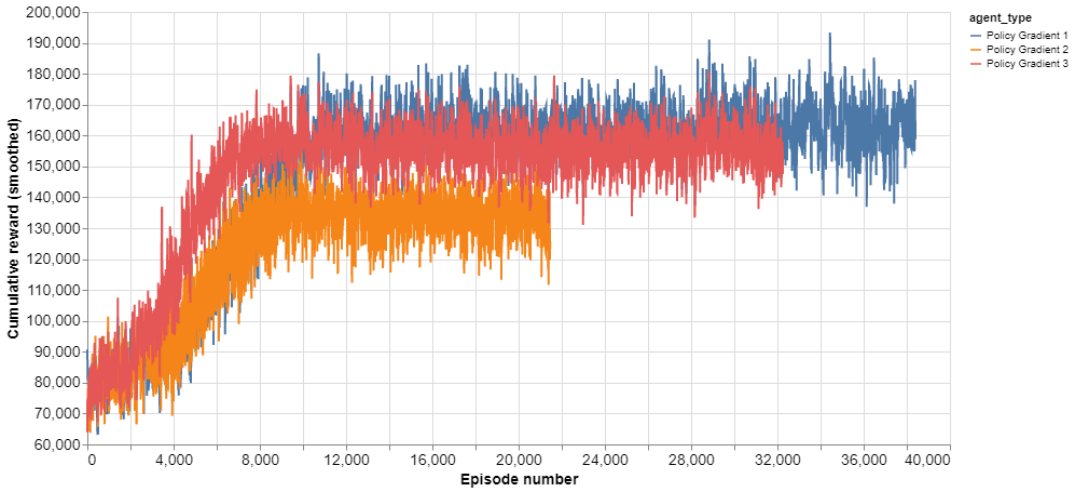


FIGURE 5.3: Learning curves of Policy Gradients agents

5.2 Agents evaluation

In order to compare different agents, 300 simulations under the same random seed were run for each type of agent. The following table shows the mean return each agent receives. As seen from the Table 5.1, RL agents outperform both a random agent and a greedy agent. Tabular Q-Learning agent has the worst performance out of all RL methods. It also requires the most time and memory resources. DQN and Policy Gradient agents have similar performance, making them a preferable choice for this task and environment.

In the Figure 5.4, one can see how much each agent receives in each episode and the overall distribution of returns of each agent.

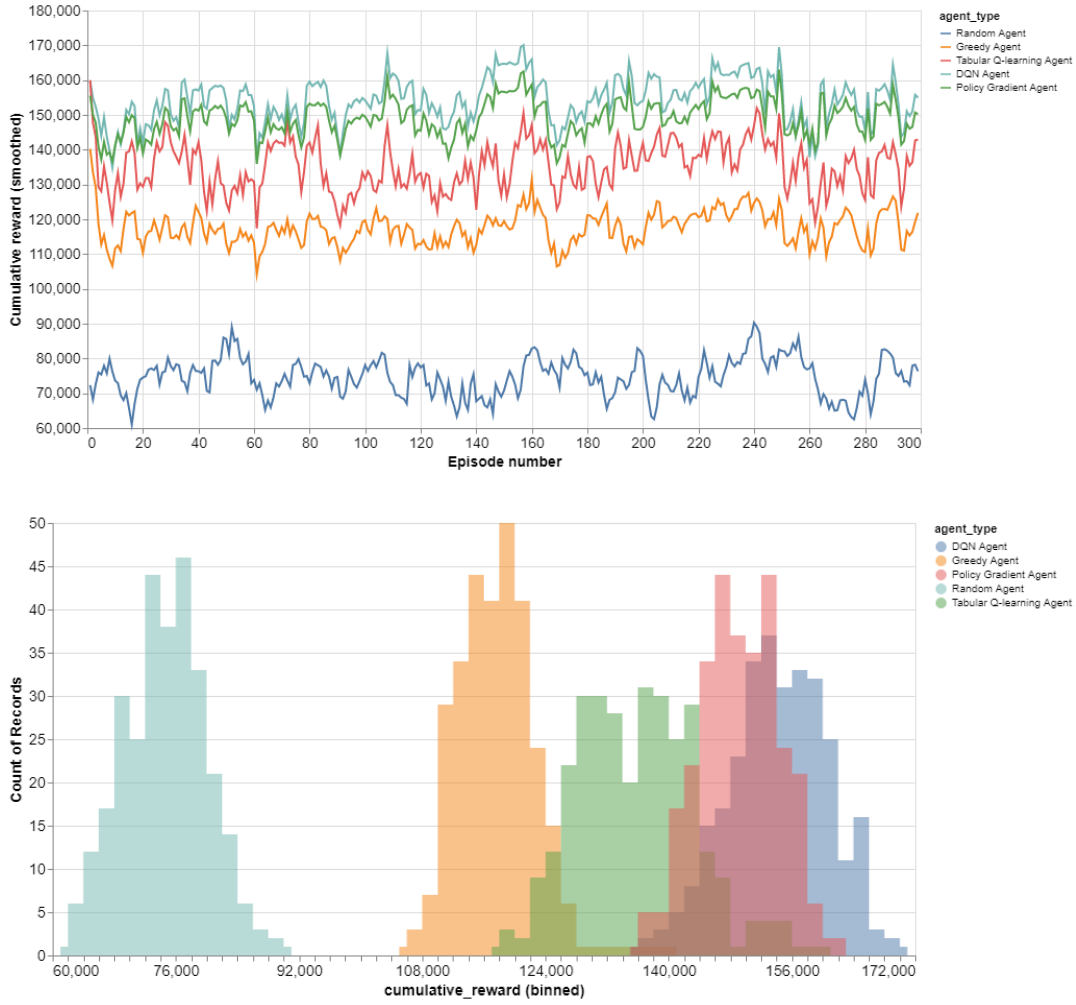


FIGURE 5.4: Performance comparison of different agents

Chapter 6

Conclusions and Future work

Dynamic pricing methods allow different businesses to reach their financial goals. Dynamic pricing methods could be implemented using ML machinery. It requires historical data to be collected and used. For this work, we have implemented a few dynamic pricing strategies for the Amazon marketplace. Customer data was used together with additional data taken from the Keepa service. The first class of methods, which is based on demand forecasting, is called greedy. They maximize the immediate revenue or other performance metrics. In our work, the random forest regressor was used to forecast the demand. Its main drawback is that it considers only current timestep revenue and could be nonoptimal in the long run. An alternative class of methods, which solve the problem of greedy decision-making, is the RL-based methods. Since training in the real market is often impractical, the simulator of the market environment was built. The problem is that it is tough to simulate the market accurately. It is the main drawback of this approach. Based on conducted experiments, we prove the effectiveness and advantages of RL methods over the greedy methods. Future works on this topic may include learning the optimal pricing strategy and simultaneously learning how much money should be spent on advertising. In the case of Amazon, it could be the PPC advertising.

Bibliography

[1] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[2] Eleks company. *2021’s E-commerce Tech Trends: Reinforcement Learning for Dynamic Pricing*. 2021. URL: https://labs.eleks.com/2021/02/e-commerce-tech-trends-reinforcement-learning-for-dynamic-pricing.html.

[3] Özlem Cosgun, Yeliz Ekinci, and Seda Yanik. “Fuzzy rule-based demand forecasting for dynamic pricing of a maritime company”. In: *Knowl. Based Syst.* 70 (2014), pp. 88–96. DOI: 10.1016/j.knosys.2014.04.015. URL: https://doi.org/10.1016/j.knosys.2014.04.015.

[4] Ludwig Dierks and Sven Seuken. “The Competitive Effects of Variance-based Pricing”. In: *CoRR* abs/2001.11769 (2020). arXiv: 2001.11769. URL: https://arxiv.org/abs/2001.11769.

[5] Marshall L. Fisher, Santiago Gallino, and Jun Li. “Competition-Based Dynamic Pricing in Online Retailing: A Methodology Validated with Field Experiments”. In: *Manag. Sci.* 64.6 (2018), pp. 2496–2514. DOI: 10.1287/mnsc.2017.2753. URL: https://doi.org/10.1287/mnsc.2017.2753.

[6] Robert F. Gök. “The Impact of Cost Based Pricing Rules on Capacity Planning Under Uncertainty”. In: 42 (2000). URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=229531.

[7] Hado van Hasselt. “Double Q-learning”. In: *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada*. Ed. by John D. Lafferty et al. Curran Associates, Inc., 2010, pp. 2613–2621. URL: https://proceedings.neurips.cc/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html.

[8] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[9] Ionel-Alexandru Hosu and Traian Rebedea. “Playing Atari Games with Deep Reinforcement Learning and Human Checkpoint Replay”. In: *CoRR* abs/1607.05077 (2016). arXiv: 1607.05077. URL: http://arxiv.org/abs/1607.05077.

[10] Shirin Joshi, Sulabh Kumra, and Ferat Sahin. “Robotic Grasping using Deep Reinforcement Learning”. In: *CoRR* abs/2007.04499 (2020). arXiv: 2007.04499. URL: https://arxiv.org/abs/2007.04499.

[11] Yaser Keneshloo, Naren Ramakrishnan, and Chandan K. Reddy. “Deep Transfer Reinforcement Learning for Text Summarization”. In: *Proceedings of the 2019 SIAM International Conference on Data Mining, SDM 2019, Calgary, Alberta, Canada, May 2-4, 2019*. Ed. by Tanya Y. Berger-Wolf and Nitesh V. Chawla.

SIAM, 2019, pp. 675–683. DOI: 10.1137/1.9781611975673.76. URL: https://doi.org/10.1137/1.9781611975673.76.

[12] George Dimitri Konidaris, Sarah Osentoski, and Philip S. Thomas. "Value Function Approximation in Reinforcement Learning Using the Fourier Basis". In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. Ed. by Wolfram Burgard and Dan Roth. AAAI Press, 2011. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3569.

[13] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. "Estimating mutual information". In: *Physical Review E* 69.6 (2004). ISSN: 1550-2376. DOI: 10.1103/physreve.69.066138. URL: http://dx.doi.org/10.1103/PhysRevE.69.066138.

[14] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2019. arXiv: 1509.02971 [cs.LG].

[15] Kyle Y. Lin. "Dynamic pricing with real-time demand learning". In: *Eur. J. Oper. Res.* 174.1 (2006), pp. 522–538. DOI: 10.1016/j.ejor.2005.01.041. URL: https://doi.org/10.1016/j.ejor.2005.01.041.

[16] Jiaxi Liu et al. "Dynamic Pricing on E-commerce Platform with Deep Reinforcement Learning". In: *CoRR* abs/1912.02572 (2019). arXiv: 1912.02572. URL: http://arxiv.org/abs/1912.02572.

[17] Jun Liu. "On the Convergence of Reinforcement Learning with Monte Carlo Exploring Starts". In: *CoRR* abs/2007.10916 (2020). arXiv: 2007.10916. URL: https://arxiv.org/abs/2007.10916.

[18] Ning Liu et al. "Deep Reinforcement Learning for Dynamic Treatment Regimes on Medical Registry Data". In: *CoRR* abs/1801.09271 (2018). arXiv: 1801.09271. URL: http://arxiv.org/abs/1801.09271.

[19] Zixia Liu et al. "A Reinforcement Learning Based Resource Management Approach for Time-critical Workloads in Distributed Computing Environment". In: *IEEE International Conference on Big Data, Big Data 2018, Seattle, WA, USA, December 10-13, 2018*. Ed. by Naoki Abe et al. IEEE, 2018, pp. 252–261. DOI: 10.1109/BigData.2018.8622393. URL: https://doi.org/10.1109/BigData.2018.8622393.

[20] Ashique Rupam Mahmood, Hado van Hasselt, and Richard S. Sutton. "Weighted importance sampling for off-policy learning with linear function approximation". In: *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. Ed. by Zoubin Ghahramani et al. 2014, pp. 3014–3022. URL: https://proceedings.neurips.cc/paper/2014/hash/be53ee61104935234b174e62a07e53cf-Abstract.html.

[21] Francisco S. Melo and M. Isabel Ribeiro. "Q -Learning with Linear Function Approximation". In: *Learning Theory, 20th Annual Conference on Learning Theory, COLT 2007, San Diego, CA, USA, June 13-15, 2007, Proceedings*. Ed. by Nader H. Bshouty and Claudio Gentile. Vol. 4539. Lecture Notes in Computer Science. Springer, 2007, pp. 308–322. DOI: 10.1007/978-3-540-72927-3_23. URL: https://doi.org/10.1007/978-3-540-72927-3_23.

[22] Teruhisa Misu et al. "Reinforcement Learning of Question-Answering Dialogue Policies for Virtual Museum Guides". In: *Proceedings of the SIGDIAL 2012 Conference, The 13th Annual Meeting of the Special Interest Group on Discourse and Dialogue, 5-6 July 2012, Seoul National University, Seoul, South Korea*. The Association for Computer Linguistics, 2012, pp. 84–93. URL: https://www.aclweb.org/anthology/W12-1611/.

[23] Elena Pashenkova, Irina Rish, and Rina Dechter. "Value iteration and policy iteration algorithms for Markov decision problem". In: 15 (1996). URL: https://www.ics.uci.edu/~dechter/publication/r42a-mdp_report.pdf.

[24] C.V.L. Raju, Y. Narahari, and K. Ravikumar. "Reinforcement learning applications in dynamic pricing of retail markets". In: *EEE International Conference on E-Commerce, 2003. CEC 2003.* 2003, pp. 339–346. DOI: 10.1109/COEC.2003.1210269.

[25] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nat.* 529.7587 (2016), pp. 484–489. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961.

[26] Richard S. Sutton. "Learning to Predict by the Methods of Temporal Differences". In: *Mach. Learn.* 3 (1988), pp. 9–44. DOI: 10.1007/BF00115009. URL: https://doi.org/10.1007/BF00115009.

[27] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998. ISBN: 978-0-262-19398-6. URL: https://www.worldcat.org/oclc/37293240.

[28] Richard S. Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*. Ed. by Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller. The MIT Press, 1999, pp. 1057–1063. URL: http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.

[29] Thibaut Théate and Damien Ernst. "An application of deep reinforcement learning to algorithmic trading". In: *Expert Syst. Appl.* 173 (2021), p. 114632. DOI: 10.1016/j.eswa.2021.114632. URL: https://doi.org/10.1016/j.eswa.2021.114632.

[30] Venishetty Sai Vineeth, Huseyin Kusetogullari, and Alain Boone. "Forecasting Sales of Truck Components: A Machine Learning Approach". In: *10th IEEE International Conference on Intelligent Systems, IS 2020, Varna, Bulgaria, August 28-30, 2020*. Ed. by Vassil Sgurev et al. IEEE, 2020, pp. 510–516. DOI: 10.1109/IS48319.2020.9200128. URL: https://doi.org/10.1109/IS48319.2020.9200128.

[31] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Mach. Learn.* 8 (1992), pp. 229–256. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696.

[32] Hiroshi Yamakawa. "Attentional Reinforcement Learning in the Brain". In: *New Gener. Comput.* 38.1 (2020), pp. 49–64. DOI: 10.1007/s00354-019-00081-z. URL: https://doi.org/10.1007/s00354-019-00081-z.

[33] Lantao Yu et al. "SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient". In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. Ed. by Satinder P. Singh and Shaul Markovitch. AAAI Press, 2017, pp. 2852–2858. URL: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14344.