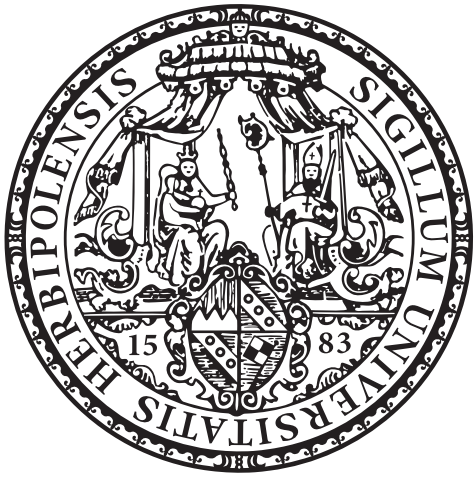
Julius-Maximilians-Universität Würzburg

Computer Science, Chair X - Data Science



Model-Free Deep Reinforcement Learning for Interactive Recommender Systems: An Evaluation

Master's Thesis in Computer Science

by

Alexander Grimm

30.04.2020

Supervisor:
Dipl.-Inf. Alexander Dallmann

Examiners:
Prof. Dr. Andreas Hotho
Prof. Dr. Frank Puppe

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Würzburg, 30. April 2020

Alexander Grimm

Abstract

Recommender Systems can be found on nearly every online platform nowadays as they have become a crucial tool for online marketing. Nevertheless, it became popular only recently to model the dynamic interaction between a user and an underlying recommendation engine as a Markov Decision Process. As Deep Reinforcement Learning methods achieve above-human performance in multiple games by now, the application of these methods to the recommendation domain is only natural. However, the proposed model-free approaches mostly suffer from a lack of comparability due to the application on proprietary datasets and varying evaluation methods.

In this thesis, several recent approaches of deep reinforcement learning based recommender systems are selected and evaluated in an explicitly defined evaluation setting. Therefore, publicly available datasets are leveraged to reinforce reproducibility of this work. Simultaneously, common pitfalls in this research area are highlighted and possible counter measures are proposed to efficiently learn from offline data. Finally, new algorithms, which have been proposed to further enhance the optimization from a fixed batch of data, are applied for the first time in the domain of recommender systems.

The results indicate that comparability is a large problem in the reinforcement learning section of recommendation research, just as it was shown recently for supervised-learning methods in this domain. The proposed performance could not be validated for the majority of approaches and it was found that all methods under consideration were more or less equal with respect to their accuracy. For this reason, the importance of using metrics that additionally measure an agent's diversity is emphasized and illustrated on exemplary movie recommendations. From the two additionally introduced approaches, which were shown to enhance offline learning in other domains, only one could achieve competitive performance.

Zusammenfassung

Empfehlungssysteme sind heutzutage auf fast jeder Online-Plattform zu finden, da sie zu einem wichtigen Werkzeug für das Online-Marketing geworden sind. Dennoch wurde es erst in jüngster Zeit populär, die dynamische Interaktion zwischen dem Benutzer und dem Algorithmus als sequenzielles Optimierungsproblem zu modellieren. Reinforcement Learning (RL) Methoden in Verbindung mit Deep Learning haben sich enorm verbessert in den letzten Jahren und in einigen Spielen bereits Menschen übertroffen. Die vorhandenen Applikationen von modell-freien RL Methoden in dem Bereich der Empfehlungssysteme lassen sich allerdings nur schwer vergleichen, da entweder verschiedene Datensätze verwendet oder unterschiedliche Evaluationsmethoden verfolgt wurden.

In dieser Arbeit werden mehrere Ansätze von ebendiesem Anwendungsbereich ausgewählt und in einem realistischen Szenario methodisch evaluiert. Gleichzeitig werden häufige Fallstricke in diesem Forschungsbereich aufgezeigt und mögliche Gegenmaßnahmen vorgeschlagen, um effektiv von gesammeltem implizitem Nutzerfeedback zu lernen. Schließlich werden zwei weitere Algorithmen evaluiert, welche in anderen Bereichen gezeigt haben, dass sie besser von Interaktionsdaten lernen können als traditionelle Verfahren.

Die Ergebnisse zeigen, dass die Vergleichbarkeit im Bereich des modellfreien Reinforcement Learnings in diesem Anwendungsgebiet ein großes Problem darstellt, so wie es kürzlich für Methoden des überwachten Lernens demonstriert wurde. Die verglichenen Algorithmen konnten ihre behauptete signifikante Verbesserung zu anderen Ansätzen nicht wiederholen und es wurde festgestellt, dass sie alle mehr oder minder gleich akkurat sind, abhängig vom gewählten Datensatz. Aus diesem Grund wird die Bedeutung von Metriken, welche die Reichhaltigkeit der Empfehlungen abbildet, hervorgehoben und an beispielhaften Filmempfehlungen veranschaulicht. Von den beiden zusätzlich angewendeten Ansätzen, die eigentlich in diesem Evaluationsszenario besser abschneiden sollten, konnte nur einer eine mit den anderen Methoden vergleichbare Leistung erzielen.

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

More series similar to Friends.
People who bought this product also bought one of these.

Netflix, Amazon

It does not matter which online service we use today, be it watching videos on YouTube, listening to songs on Spotify or shopping on Amazon - as soon as one opens the homepage, he is immediately confronted with recommendations for what to consume next. This makes sense as it is the company's goal to keep its users engaged in order to keep them as a client and thereby increase profit. As the number of offered products increases to the millions, consumers are quickly overwhelmed by the selection and need to be presented a carefully picked subset, personalized to their taste. Good recommendation systems were shown to be a crucial tool of any internet platform. For example, already in 2013, 35% of products purchased on Amazon and 75% of what people watched on Netflix were introduced through such systems [1]. In 2018, YouTube's product chief officer mentioned that 70% of the time users watch a video, they where recommended by their algorithm [2].

Many methods proposed in the field's research community model the recommendation process as a static prediction method, e.g. given the user just bought one article, what is the most likely one he will purchase next? While this approach typically achieves good user satisfaction, it is probably not the optimal approach from an economical point of view. From a business

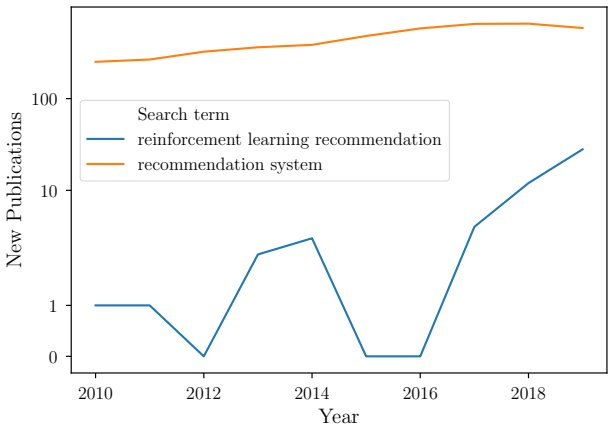


Figure 1.1: Number of publications added per year, note the logarithmic scale of the y-axis. The data was collected on Google Scholar on 14th of March 2020.

perspective, it is way more profitable to recommend products that will result in more transactions in the future. For example, a user is browsing for a book on Amazon and a Recommender System (RS) shall recommend suitable products. Given the choice between a novel with 30% chance that the user will buy it and a second one with only 25% chance, which however is the first part of a series, a static recommendation method will always choose the first one. However, in the long run, it will probably be more profitable to recommend the second book because if the consumer liked it, he will most likely purchase the rest of the series as well [3].
For this reason it is a more reasonable approach to see the interaction between a user and the RS as a sequential optimization problem and therefore model it as a Markov Decision Process (MDP) [3]. The user makes a decision for his next interaction based on another one in the past, e.g. buying that second book in the series. Reinforcement Learning (RL) is the typical approach to find an optimal policy for a MDP. Especially Deep Reinforcement Learning (DRL) has attracted a lot of attention in the last years. Google's AlphaGo [4] won against human world champions in Go and most recently AlphaStar [5] beat one of the best professional players in Starcraft II. So why shouldn't it be possible to create the best "player" that can sell the most products to users through personalized recommendations?

In practice, applying DRL in the domain of recommender systems has become a popular approach only recently. Figure 1.1 shows the number of publications over the last ten years for DRL-based and general recommendation systems. The actual number of publications per year here is less

important than the clearly observable trend to model the recommendation problem as a MDP.

However, creating good recommendation methods comes with many challenges and, most importantly, the availability of data. For this reason, many new algorithms were published by large companies such as YouTube [6] or JD [7], showing performance gains merely on their individual platform. Secondly, many researchers used different evaluation procedures or task settings, which prohibits explicit comparability between the approaches. Popular baselines for evaluation are often bandit algorithms and even pure random choices in some cases. One of the reasons, of course, was that many publications approximately appeared at the same time, as indicated by Fig. 1.1.

The aim of this work is to evaluate and compare popular approaches of DRL-based recommendation methods fairly. Therefore, a common task is chosen where a user directly interacts with a recommender system and data is given by logs of implicit user feedback trajectories. The study focuses on model-free approaches, a sub-category of RL, where an agent has got no internal world model for explicit planning but only learns from interactions with an environment. In the process, typical challenges of applying DRL to this real world task are highlighted and solutions are elaborated to contain some of them. In addition to that, the applied evaluation and data processing procedures are described in detail to reinforce the reproducibility of this research. Finally, most recent methods that enhance full off-policy learning in other domains are applied in the recommendation domain for the first time.

This evaluation demonstrates that none of the approaches under consideration is consistently better than others over all datasets, as some claim in their original publication. One approach was even unable to learn a policy at all. Furthermore, all RL agents were outperformed by a typical supervised-learning baseline. A distributional RL-agent and an ensemble version of a Deep Q-Network (DQN), which have been shown to improve off-policy learning from batch in other works [8], do not work significantly better than other approaches.
Along the way, the methodology of learning from collected user trajectories is outlined and it is highlighted that negative examples are crucial for efficient offline optimization of value-based algorithms. Finally, detailed insights of inference behavior emphasize that RS should not exclusively be judged by classification metrics.

This thesis is organized as follows. At first, preliminary information is provided with a deep introduction to Reinforcement Learning and a brief glance into function approximation methods. Chapter 3 provides an overview of related work in recommender systems with a special focus on methods us-

ing DRL. The methodology is described in Chapter 4. Here, the MDP and the recommendation task is defined in a formal manner and open challenges of the productive application of DRL methods are highlighted. The chapter continues with an introduction of the compared algorithms and concludes with the evaluation strategy. Chapter 5 contains information about the experimental setup and the results of the evaluation. This thesis concludes with a discussion of the results in Chapter 6 and an outlook for future work.

2

Preliminaries

This chapter provides an introduction to the concepts involved in Deep Reinforcement Learning. At first RL will be introduced in comprehensible detail. As problems become more complex, classic approaches quickly become unfeasible and therefore need to leverage function approximation. For this, Deep Learning algorithms are used, which will be covered briefly at the end of this chapter.

2.1 Reinforcement Learning

It is most common to model sequential optimization problems as a Markov Decision Process. Accordingly, it will be defined generally in the beginning of this section. Reinforcement Learning is the method of choice in this work to find an optimal solution for a MDP, where an agent interacts with its environment repeatedly until convergence. For this reason, key terminology of this field will be introduced first, followed by the general iterative procedure for successive policy optimization. Lastly, the three main families of RL approaches are described and their respective advantages and inconveniences are highlighted.

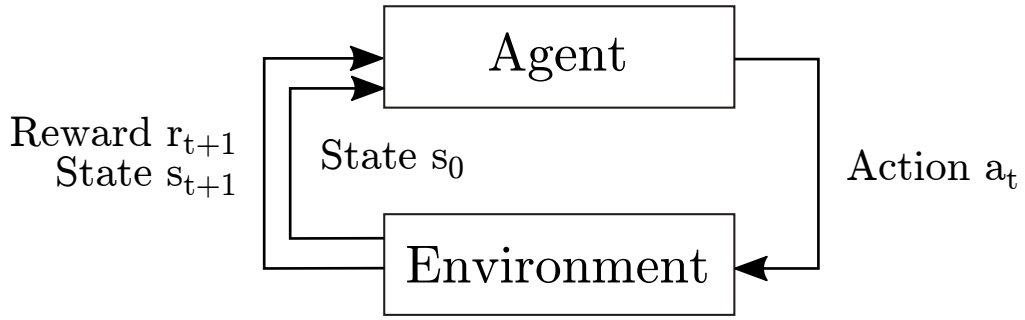


Figure 2.1: The interaction feedback loop in Reinforcement Learning. Given an initial state s_0 from the environment, the agent computes an appropriate action to execute. Then, the environment provides feedback for this action, the reward, together with the next state which is the result of taking the chosen action in the previous state.

2.1.1 Markov Decision Process

A Markov Decision Process (MDP) is a discrete time stochastic process that models the interaction between a decision maker, who will be called an agent henceforth, and an environment [9]. At any time, an agent is in a state and can transition to another one by executing an action in the environment. The agent then receives a reward for selecting this action, and continues to traverse between states with the goal to maximize the sum of received rewards. The general procedure is illustrated in Fig. 2.1 and the MDP can be defined formally as a quintuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ as follows:

- **States** $\mathcal{S}$: A set of all possible states an agent can be in. A state can be of any form, from a discrete scalar value to a continuous vector.
- **Actions** $\mathcal{A}$: A set of all actions the agent can choose. An action can also be continuous or discrete.
- **Transition Probabilities** $\boldsymbol{P}$: A probabilistic function $P : S \times A \to S$ which models the transitions between states for a given action. In most cases, this function is unknown and will be estimated by the agent through interaction with the environment.
- **Reward function** R: A reward function $R : S \times A \to \mathbb{R}$ that values the transition from state s_t to s_{t+1}, based on the executed action a_t.
- **Discount factor** $\boldsymbol{\gamma}$: A constant weight $\gamma \in [0, 1]$ that determines the trade-off between a myopic and a long-term strategy. A policy with

$\gamma = 1$ takes all future rewards equally into account while choosing a value close to 0 optimizes for an immediate reward.

The foundation of a MDP is given by the Markov Property. This means, the transition probability from the current state to the next one in a sequence of transitions only depends on the current state:

$$P(s_{t+1} \mid s_1, \ldots, s_t) = P(s_{t+1} \mid s_t) \; . \tag{2.1}$$

In this work, the notation from Sutton and Barto [10] is followed that describe the environment dynamics by merely one function. Let $s, s' \in \mathcal{S}$, $a \in \mathcal{A}$ and $r \in \mathcal{R}$ be random variables for two consecutive states, an action and a reward respectively. Then, one can define a dynamics function of the finite MDP as

$$p(s', r \mid s, a) = Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \; . \tag{2.2}$$

This equation describes all state transitions with their respective rewards in the environment. However, other sources [9, 11] do not describe the MDP by only one function but define the transition probability function P directly. Nevertheless, given Eq. (2.2) one can recover the state-transition probabilities by

$$P(s' \mid s, a) = \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \tag{2.3}$$

or even compute the immediate expected reward

$$r(s, a) = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a) \; . \tag{2.4}$$

Equation (2.3) describes the probability that the environment will traverse to state s' when choosing action a in the current state s.

2.1.2 Main Components

In the following, the main components and notations are introduced which are used throughout the rest of this work.

Return. The main goal in Reinforcement Learning is to maximize the expected long-term return. In particular, given a trajectory of interactions $\tau = [s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T]$, the *return* at time t is the sum of rewards

over all time steps:

$$\begin{aligned} G_t(\tau) &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \\ &= r_{t+1} + \gamma G_{t+1}(\tau_{t+1:T}) \quad . \end{aligned} \tag{2.5}$$

In addition, the return is usually discounted by a *discount factor* $\gamma \in]0, 1]$ so that immediate rewards are more important than those in the future. Further, it guarantees convergence of the return series if the trajectory is very long or even infinite for $\gamma < 1$ [10]. Henceforth, the *immediate reward* at time t will be notated r_t which is the result of evaluating the reward function $R(s_t, a_t)$, whereas the *return* is G_t. In Equation (2.5) the infinite-horizon notation is used, where the trajectory has infinite length. In practice though, the discounted expected return is also applied for finite-horizon trajectories with length T, where T can be seen as another random variable, because it was found to be beneficial in practice [10].

Policy. A *policy* is a decision function which chooses an action given the current state. In case of deterministic policies one notates $a_t = \mu(s_t)$ while for stochastic policies one samples from the policy, which resembles a distribution over all actions, conditioned on the state:

$$a_t \sim \pi(\cdot \mid s_t) \quad .$$

When referring to *policy* in RL, it is mostly used exchangeable with *agent* as the behavior in a given state. Later in this work, the term *actor* will be introduced as well. All three terms refer to the same entity, i.e. the decision maker in RL. Here, the policy is always notated by π for an unambiguous notation, no matter if a deterministic or a stochastic policy is followed.

Value Functions. So, how does a policy decide which is the best action to take in the current state? Here, the quality metric for a state is the highest estimated return that can be obtained continuing from that state. This metric is approximated by the *value function*, which maps a state to the return that can be expected henceforth from that state. In particular, a value function is always subject to the given policy π:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[G_t \mid S_t = s\right] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \;\middle|\; S_t = s\right], \forall s \in \mathcal{S} \; . \tag{2.6}$$

More specifically, the value of a given state s is the expected return from starting in that particular state at time t and continuing to follow policy π henceforth. In finite horizon settings, i.e. an episode ends with a terminal state at some point, the return of the final state is defined to be zero.
The value function describes which state is more promising than others but it still does not describe how to get to this state. Therefore, one defines an *action-value function* Q as the expected return of starting from s, selecting action a under the given policy π and continuing to follow that policy:

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}_{\pi}\left[G_t \mid S_t = s, A_t = a\right] \\ &= \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, S_t = s, A_t = a\right] . \end{aligned} \tag{2.7}$$

The action-value function is usually referred to as the *Q-Function* as it measures the *quality* of an action in a given state. One observes that both value functions are merely expectations of the return while following policy π. In the classical tabular reinforcement learning setting, one could compute values for each state through Dynamic Programming (DP). DP is a mathematical method to efficiently solve a complex problem by recursively solving subproblems, introduced by Bellman [12]. In particular, after a full episode of a game, one calculates all intermediate action-values for the trajectory path through backtracking. All values are then saved in a table for each state-action pair respectively. After enough samples of all possible state-action combinations have been collected, the mean values converge to the actual expected return.
However, tracking all possible combinations is intractable in practice due to the typical large number of states and actions. Consequently, the value functions are usually approximated by parameterized functions, which will be covered in Section 2.2.

Bellman equations. Just like the return function in Eq. (2.5), the value-functions can be written in a recursive representation. The value function V^{π}, as defined in Eq. (2.6) can thus be extended to

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}\left[G_t \mid S_t = s\right] \\ &= \mathbb{E}_{\pi}\left[r_t + \gamma G_{t+1} \mid S_t = s\right] \\ &= \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{r \in \mathcal{R}, s' \in \mathcal{S}} p(s', r \mid s, a)\left[r + \gamma V^{\pi}(s')\right] . \end{aligned} \tag{2.8}$$

In other words, the expected value of a state s is the immediate reward plus the expected value of each reachable next state. Most importantly, the state values are weighted by their respective probability of actually being in that state, expressed through the dynamics function of the environment (Eq. (2.2)). Equation (2.8) is called the *Bellman equation* with the unique solution V^π.

In a similar manner, one can derive the *Bellman equation* for Q^π

$$Q^\pi(s,a) = \sum_{r\in\mathcal{R},s'\in\mathcal{S}} p(s',r \mid s,a)\left[r + \gamma \sum_{a'\in\mathcal{A}} \pi(a' \mid s')Q^\pi(s',a')\right] . \tag{2.9}$$

These equations lay the basics for iterative algorithms to find the optimal policy over a discrete time sequence.

Optimal functions. In Reinforcement Learning one wants to find a policy π which gets the highest possible accumulated reward along a trajectory. Many different policies can be learned to magnify the return, yet some policies are better than others. Therefore, a policy π is said to be better than π' if $V_\pi(s) \geq V_{\pi'}(s), \forall s \in \mathcal{S}$, i.e. a higher return is expected in each state. Following this ordering of policies, there is always at least one that is better than the others. This is the *optimal policy*

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}_{\tau\sim\pi}\left[G(\tau)\right] \tag{2.10}$$

which expects the highest return in all trajectories τ.
All optimal policies share the same *optimal value functions* and the optimal action-value and value function can be expressed in terms of each other:

$$Q^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma V^*(S_{t+1}) \mid S_t = s, A_t = a\right] \tag{2.11}$$

because the Q-function gives the expected return for taking the action a in s and then follows the optimal policy.

Equivalently, one can express the optimal value-function through Q by following the intuition, that the optimal state value must be equal to the return of executing the best action. This is then called the *Bellman optimality equation* for the optimal state value

$$
\begin{aligned}
V^*(s) &= \max_a Q_{\pi^*}(s,a) \\
&= \max_a \mathbb{E}\left[R_{t+1} + \gamma V^*(S_{t+1}) \mid S_t = s, A_t = a\right] \qquad (by\ Eq.\ (2.11)) \\
&= \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V^*(s')\right] \qquad (by\ Eq.\ (2.8))
\end{aligned}
$$

and the optimal action-value

$$
\begin{aligned}
Q^*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') \mid S_t = s, A_t = a\right] \\
&= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} Q^*(s', a')\right] .
\end{aligned}
$$

It can be observed that both equations are independent of the actual policy π and thus represent an equation system with one equation for each state. In order to calculate an exact solution with any algorithm that solves non-linear equations, three assumptions must hold: 1.) The state and action spaces need to be small, 2.) the dynamics p of the MDP must be known and 3.) the environment actually follows the Markov Property (Eq. (2.1)). All three of those aspects need to be fulfilled to compute an *exact* optimal solution for the Reinforcement Learning problem, which is to find an optimal value function that acts with an optimal policy. In practice though, at least one of the assumptions is typically not given and thus the solution for the optimal equations must be approximated through an iterative optimization procedure, which will be covered next.

2.1.3 Iterative Policy Optimization

In this section, the general technique of approximating the optimal policy and value-functions for solving Markov Decision Problems is described. At first, some key-words for distinguishing different methods are introduced before presenting the iterative procedure for a solution that is followed by practically all approaches.

Method distinctions

There are different techniques to conduct a solution for the stated MDP problem. The approaches can be separated into two groups: 1.) algorithms which use or build an explicit model of the environment and 2.) model-free approaches. In this work, only the model-free methods will be further

investigated because one usually wants to use an algorithm which can solve many problems without the need of specific domain knowledge. In particular, model-free methods have to approximate the environment's dynamics function (Eq. (2.2)) through interaction with it, while model-based algorithms have a world model available or build it along the learning process.
Algorithms can further be distinguished in *on-policy* or *off-policy*. This is done by separating the policy into two concepts: a behavior and a target policy. The behavior policy directly interacts with the environment and is responsible for collecting experience. On the other hand, the target policy is the one that is being learned and becomes the optimal policy eventually. *Off-policy* methods stand out due to the ability of optimizing a target-policy from experience samples collected by another (behavior) policy. In practice, this means that off-policy methods can reuse transition data that was generated from previous versions of the current policy or even a totally different agent. Contrarily, *on-policy* methods must use interactions which were collected by the current policy because the target and behavior policy are identical.
Consequently, off-policy algorithms tend to have higher variance, as they improve by following actions chosen by another policy. However, this inconvenience is only a trade-off for a higher sample-efficiency as more experience can be used for policy updates. On-policy approaches, on the other hand, converge much slower but therefore smoother towards the optimal policy due to the more stable updates. [10]

General Procedure: Policy Iteration

As a start into RL methods, it is handy to know the basic functionality of almost all algorithms. In principle, the learning of an optimal policy and value-function follows an iterative and competing process, which is illustrated in Algorithm 1. At first (line 6–14), the value-function is updated to create better approximations for the value of each state when following the current policy π. This process is referred to as *policy evaluation*. On the other hand, at *policy improvement* step, the policy is updated to act greedy w.r.t the value-function (line 16–25). Here the roles are reversed, i.e. the policy optimizes itself to act better with respect to the next expected values, approximated by the current V^π. When repeating these two steps in an iterative manner, π and V^π converge in the limit to the optimal version of both, hence to π^* and V^*. This iterative process is called Generalized Policy Iteration (GPI) and is the basic principle for all further descriptions in this section. Note that the algorithm describes the case of a dynamic programming scenario where state and action spaces are of manageable size. For larger scenarios the two blocks update alternately per time-step and optimize

Algorithm 1 Policy Iteration in Dynamic Programming [10].

procedure POLICY-ITERATION(convergence-threshold κ)
    # Initialization
    Initialize $V(s)$ and $\pi(s)$ arbitrarily for all $s \in S$

    while $t < T$ **do**
        # Policy Evaluation / Prediction
        repeat
            $\nabla = 0$
            for all $s \in S$ **do**
                $v \leftarrow V(s)$
                $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s)) \left[R_{t+1} + \gamma V(s')\right]$
                $\nabla \leftarrow max(\nabla, |v - V(s)|)$
            end for
        until $\nabla < \kappa$

        # Policy Improvement / Control
        $stable \leftarrow true$
        for all $s \in S$ **do**
            $a \leftarrow \pi(s)$
            $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r \mid s, a) \left[R_{t+1} + \gamma V(s')\right]$
            $stable \leftarrow stable \wedge (a == \pi(s))$
        end for
        if stable **then**
            return V, π
        end if
    end while
end procedure

over many trajectories.

Exploration vs. Exploitation. One of the main dilemmas in Reinforcement Learning is the trade-off between exploration and exploitation. On the one hand, an agent should follow its experience and act greedily according to empirical rewards. For example, for a given state s, the action-value function evaluates to a clear maximum action a. Hence, the agent should exploit its already acquired knowledge. On the other hand, one wants the agent to explore a variety of actions for a state, to ensure the agent has at least tried taking another way, even though it might not seem optimal at that point. Only through exploration, the policy can find better actions over time for all states and can become the true optimal policy.
Exploration can be enforced in different ways. In continuous action-spaces, one usually adds noise to the policy's action before executing it in the environment [13]. In discrete action-spaces it depends on the kind of the actor -

stochastic or deterministic. Stochastic actors automatically introduce some exploration by sampling from the distribution instead of choosing the greedy action. Deterministic actors typically follow an ϵ-greedy policy for prediction. This means, that with a probability of $(1 - \epsilon)$ the actor follows its greedy policy, while with a probability of ϵ it selects any action from the available action space by random [14, 15].

2.1.4 Temporal Difference Learning

At first, the general notation and intuition for Temporal-Difference (TD) Learning will be outlined before introducing the well known Q-Learning algorithm.
Earlier, it was briefly mentioned that the MDP-problem can be solved through DP. As stated previously, this method is only feasible if the full dynamics function of the environment is known. As this is practically never given, a first approximation technique, namely Monte Carlo (MC) methods can be applied to approximate the value function instead. They do not need access to the environment's dynamics but they do have to interact with the environment until the end of an episode before an update of the value function can be applied. One episode is the interaction between agent and environment until a terminal state is reached.
This is where TD-Learning closes the gap. It is a combination of both approaches, i.e. it can learn directly from experience like MC-methods and updates its value function through *bootstrapping*, like a DP-approach. MC-methods are known to have a high variance, as the estimation errors propagate until the end of an episode. TD, in contrast, does not wait for an episode to finish in order to obtain the real return G_t but updates its value-function after every time-step according to the bootstrap estimation $R_{t+1} + \gamma V(s')$. The most basic Temporal-Difference method is the one-step TD-update

$$V(S_t) \leftarrow V(S_t) + \alpha \left[(r_{t+1} + \gamma V(S_{t+1})) - V(S_t) \right] \tag{2.12}$$

which can be calculated immediately after the transition between time steps, receiving the immediate reward. The parameter α is a positive learning rate that smooths the changes in the value function. As the term in square brackets is an error which represents the quality of the estimation, it is commonly known as the Temporal-Difference Error (TD-Error) $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$. The first part of the term was already encountered in the Bellman equation of V^π. In the update of Eq. (2.12) though, the return at this time step is a sample estimate in contrast to the expectation in Eq. (2.8) where it was directly computed over the given transition probabilities. In addition, an

Algorithm 2 Q-Learning [10]

procedure Q-Learning
  Arbitrarily initialize $Q(s, a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}$
  Set terminal states $Q(terminal, \cdot) = 0$
  for all episodes **do**
    $s \leftarrow \mathcal{S}$ from environment
    for all steps t in episode **do**
      $a \leftarrow \text{argmax}_a\, Q(s, a)$ or ϵ-greedy ▷ Improvement
      $s', r \sim P(s', r \mid s, a)$ from environment
      Update Q by Eq. (2.13) with $S_{t+1} = s'$ ▷ Evaluation
      $s \leftarrow s'$
    end for
  end for
end procedure

approximate value-function $V(s)$ is used in contrast to the true value-function V^π. Thus, the TD function updates the state-value over an estimate of the future return and V itself. Hence, TD-Learning is a bootstrapping method, just like Dynamic Programming. Note that the update in Eq. (2.12) would be the equivalent to *Policy Prediction* step in GPI from above. *Policy Control* would be to greedily follow the updated value function.

In conclusion, TD-Learning manages to combine the advantages of DP and MC methods respectively, while also solving its inconveniences. While this might sound to good to actually work in practice, it was even proven to converge to the true value function V^π under some assumptions by Dayan [16].

Similar to the introduction section to MDP, the TD-Learning technique with a state-value function can be extended to the action-value function as well. The resulting method is called *Q-Learning* [17] and is defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \quad (2.13)$$

where α is the learning rate. The function Q is proven to directly approximate the optimal action-value function Q^*, independent of the behavior policy [17]. Hence, it is an *off-policy* algorithm which can learn the optimal policy from data collected by another policy b. The only requirement for the policy to be learned efficiently is, that all states and actions are observed frequently enough. As this is one of the main techniques used in RL, the method is described in Algorithm 2.

In Eq. (2.13), the Q-Function uses the maximum of its value estimates as an estimate of the maximum of the true values. This introduces bias to

the action value, that is often referred to as *overestimation bias*. Double Q-Learning [18] tackles this issue by introducing a second Q-function. In particular, the first function selects the best action according to its maximum value estimate but the respective *value* for the TD-Error computation is evaluated by a second Q-function. This only requires a minimal change of Eq. (2.13)

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_2(S_{t+1}, a') - Q_1(S_t, A_t)\right] \quad (2.14)$$
$$\text{where } a' = \underset{a}{\operatorname{argmax}}\, Q_1\left(S_{t+1}, a\right)$$

but it leads to less bias and a better convergence property. The two Q-functions are typically optimized alternately or in a random fashion and the update of Q_2 follows equivalently with switched indices. It is observable that each Q-function actually has seen less state transitions at a time t than the one function in the original algorithm, because each one is only updated every e.g. second interaction. The introduced stability, however, increases the achievable return values drastically [18]. This form of overestimation correction is applied in nearly all current approaches, involving the ones considered in the remainder of this work.

2.1.5 Policy Gradient

In the previous introduction to RL, there was always a value function involved, either the state-value function V or the action-value function Q. Yet, there exists another family of algorithms which find the optimal policy π without any of those. Policy Gradient (PG) methods aim to find an optimal parameterization θ of a policy function $\pi(a \mid s, \theta)$ to solve the MDP. Let $J(\theta)$ be a performance measure with respect to parameters θ. Then, a Policy Gradient method maximizes this measure by following its gradient

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)} \,, \quad (2.15)$$

where $\widehat{\nabla J}$ is an estimation of the gradients that maximize J. The only requirement for this approach is that $\pi(a \mid s, \theta)$ is differentiable w.r.t θ. Let $h(s, a, \theta) \in \mathbb{R}$ be a function which calculates a numerical preference for taking a in s, parameterized by θ. Then an exponential-softmax distribution can be obtained by

$$\pi_\theta(a \mid s) = \frac{e^{h(s,a,\theta)}}{\sum_{a'} e^{h(s,a',\theta)}} \,. \quad (2.16)$$

The preference function h can be given e.g. by a neural network or any other classifier algorithm. As one can sample from that distribution, the policy gradient method already reinforces exploration. Another advantage to value methods is that the policy function may be a simpler function to approximate. Further, the policy optimization updates its parameters smoothly according to the gradient, which leads to stronger convergence guarantees and less variance. [10]
REINFORCE [19] was the first algorithm which directly optimized a policy through the policy gradient. In its procedure it is still the main PG-method today, although some improvements were introduced for higher efficiency in the meantime [20]. The following description is taken from Sugiyama, Hachiya, and Morimura [11]. Let $\tau = [s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T]$ be a trajectory of length T obtained by following a policy π_θ. As before, the objective is to maximize the return for all trajectories. Hence, one chooses the expected discounted return

$$J(\theta) = \mathbb{E}_{p(\tau|\theta)}\left[G(\tau)\right] = \int p(\tau \mid \theta) G(\tau) \mathrm{d}\tau \ , \tag{2.17}$$

as a quality metric of the policy, where $p(\tau \mid \theta)$ is the probability density of observing this trajectory under the current policy π_θ, i.e.

$$p(\tau \mid \theta) = p(s_0) \prod_{t=1}^{T} p(s_{t+1} \mid s_t, a_t) \pi_\theta(a_t \mid s_t) \ . \tag{2.18}$$

Note that the expectation is only valid under the current parameters. This means, that Policy Gradient is an *on-policy* procedure.

The parameter update is performed by Eq. (2.15) while now the gradient of the objective function Eq. (2.17) is defined as

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \int p(\tau \mid \theta) G(\tau) \ \mathrm{d}\tau \\
&= \int \nabla_\theta p(\tau \mid \theta) G(\tau) \ \mathrm{d}\tau \\
&= \int p(\tau \mid \theta) \nabla_\theta \log p(\tau \mid \theta) G(\tau) \ \mathrm{d}\tau \qquad \text{(log-deriv-trick)} \\
&= \int p(\tau \mid \theta) \nabla_\theta \sum_{t=0}^{T} \log \pi_\theta \left(a_t \mid s_t\right) G(\tau) \ \mathrm{d}\tau \\
&= \mathbb{E}_{p(\tau|\theta)} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t\right) G(\tau) \right] \ .
\end{aligned}$$

In the third line, the conversion is based on the derivation of the logarithm $\nabla log(f(x)) = \frac{\nabla f(x)}{f(x)}$ and its respective rearrangement. Applying the logarithm to Eq. (2.18) converts the product over all time steps into a sum. Calculating the derivative with respect to θ, eliminates the gradients of the environment's observation probabilities. The last equation is merely expectation, hence it can be approximated by a sample mean. Given a set of trajectories D, the gradient can be estimated by

$$\widehat{\nabla_\theta J(\theta)} = \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t\right) G(\tau) \ . \tag{2.19}$$

In conclusion, if the policy function π_θ is differentiable w.r.t θ and experience can be collected by interacting with the environment, the policy gradient can be estimated which allows an optimized parameter update. Further, Policy Gradient methods can naturally handle continuous actions and are the preferred approach when dealing with large action spaces [10].

In practice though, the gradients tend to be large, which leads to slow and unstable convergence. This is typically eliminated by introducing a *baseline* which reduces the variance of the gradient estimate. The general form of the policy gradient with baseline is

$$\nabla_\theta J(\theta) = \mathbb{E}_{p(\tau|\theta)} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t\right) \cdot \Phi \right] \ . \tag{2.20}$$

A baseline method can be any function, as long as it does not vary with the action a. The most simple one would be to choose the return over the whole trajectory, as used above in Eq. (2.19). As this also includes the past (the return G is a constant sum over the whole trajectory), a reward-to-go method $G_{t:T}$ is already a preferred choice over the previous one. Other forms are available, too, for example using the *Advantage Function* $A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$ which expresses the relative superiority of one action over the others when following π_θ. [10]

Finally, the parameters θ of π are generally updated by

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \log \pi \left(a_t \mid s_t, \theta\right) \Phi_t \ .$$

As the policy gradient is the expectation over the probability-weighted gradients, one can use a sample at every time step to approximate $\widehat{\nabla_\theta J(\theta)}$. Note that in this section, the *undiscounted* return case was discussed. Nevertheless, it is straightforward to expand it to a discounted MDP with $\gamma < 1$. The classic REINFORCE version with a discount factor is then

$$\theta_{t+1} = \theta_t + \alpha\gamma^t \nabla_\theta \log \pi_\theta (a_t \mid s_t) \cdot G_t \ .$$

2.1.6 Actor-Critic

Before the introduction of the last popular approach, it is important to recapitulate the strengths and weaknesses of the previous two methods.
Temporal Difference methods like Q-learning can estimate a value function through bootstrapping after each time step. Furthermore, Q-Learning can approximately learn the optimal value function without any involvement of a separate optimal policy. For its evaluation of the best next action, the action-value needs to be computed for all possibilities in a state. This limits the naive application of this method to a manageable size of actions and thereby explicitly prohibits continuous actions.
Policy Gradient methods on the other hand typically involve a Monte Carlo estimation of the return and hence tend to have high variance and consequently learn slowly. Through introduction of an unbiased baseline factor like the advantage function, this noise can be reduced. The main intuition behind this is to weight the action gradients with their advantage over other states, in contrast to the pure average return [21]. Following the approach of gradient ascent, the convergence is usually smoother than the one in TD-methods, as those tend to change the action-value function more drastically when following an ϵ-greedy behavior policy. Contrarily, Policy Gradient methods typically need a lot of experience trajectories in order to reach its optimum. This is no problem, when the actor can interact with a simulation environment infinitely often, but might lead to restrictions if this is not an option. By only optimizing the policy, and no action-value function, PG-methods can handle much larger action-spaces and also continuous actions, as the policy is just another function of state.

In order to get the best of both techniques, the Actor-Critic was proposed [22]. It is an interplay between an actor (policy) which is optimized by the feedback of a critic (value-function). Formally, the critic can be implemented into the policy gradient method as a *bootstrapping* baseline

$$\begin{aligned}
\theta_{t+1} &= \theta_t + \alpha \nabla_\theta \log \pi_\theta (a_t \mid s_t) \Phi_t \\
&= \theta_t + \alpha \nabla_\theta \log \pi_\theta (a_t \mid s_t)(R_{t+1} + \gamma^t V(s_{t+1}) - V(s)) \\
&= \theta_t + \alpha \delta_t \nabla_\theta \log \pi_\theta (a_t \mid s_t) \ . \qquad (2.21)
\end{aligned}$$

The TD-Error δ_t functions as a weight for the action probability gradients. This introduces some desirable bias through bootstrapping from TD-Learning and simultaneously leads to a reduction of the variance.

In REINFORCE-with-baseline, the value-function is also a typical choice as a baseline, as mentioned before. This parameterized function V_ω is learned along the process of updating the policy parameters. In contrast to Actor-Critic though, it is not used for bootstrapping there. In particular, in REINFORCE a baseline $\Phi_t = G - \hat{V}(s_t, \omega)$ backtracks the *actual* return of the recently finished trajectory and subtracts the estimated value of that state. Actor-Critic on the other hand *bootstraps* the one-step return over the TD-Error.

2.2 Function Approximation

In the previous section, it was mentioned in numerous places that exact solutions for the value-function are not feasible in practice. This is typically due to a large state space or a large or continuous action space. Therefore, the function approximation methods which are most common these days, neural networks, will be introduced in the following.

2.2.1 Feedforward Networks

In general, function approximation (or estimation) predicts a mapping $f(x)$ from an input vector x to a target variable y, i.e. $y = f(x) + \beta$ [23]. Here, β is a bias that is not predictable from the input vector. Feedforward neural networks, or Multilayer Perceptrons (MLPs), made a major contribution to function approximation methods [24]. Specifically, a MLP defines a mapping $y = f(x, \theta)$ by learning those parameters θ that lead to the best possible approximation of y for a set of given input vectors $\vec{x}$. Typically, $f(\vec{x})$ consists of a chain of functions where each function has the form

$$g(\vec{x}) = \sigma\left(W\vec{x} + \vec{b}\right)$$

where W is a learnable weight matrix and $\vec{b}$ is an optional bias vector. The activation function σ is substantial to approximate non-linear dependencies. Further terminology will be introduced on an exemplary MLP:

$$\begin{aligned} f(x) &= g\left(h\left(j\left(\vec{x}\right)\right)\right) \\ &= \sigma_{out}\left(W\sigma_h\left(U\sigma_h\left(V\vec{x} + \vec{b_3}\right) + \vec{b_2}\right) + \vec{b_1}\right) \ . \end{aligned}$$

Each individual function is called a *layer*, and all chained functions together form the *network*. The first function, here $j(\vec{x})$, is called the *input layer*, the

last function $g(\vec{z})$ the *output layer*, and everything between are *hidden layers*. MLP can solve non-linear problems by leveraging non-linear activation functions after each layer. A typical choice is the ReLU [25] activation, which also has more desirable properties for training but other non-linearities such as *tanh* or *sigmoid* functions are also valid and commonly seen choices. The output function is chosen based on the desired use case. Binary classifiers typically use the *sigmoid*, Multi-Class estimators leverage a softmax-function, as in Eq. (2.16), and a regression model could use *tanh* to restrict the values to a specified interval.
The parameters of the network, i.e. all weight matrices and bias vectors, are learned through backpropagation of the prediction error, measured by a differentiable loss function. Therefore, one iteratively updates the parameters through a gradient descent method of choice to minimize the given objective [23].

2.2.2 Sequential Approximators

While the classic MLP has great success on independent data points, it becomes sub-optimal when data follows a sequential pattern such as time-series or natural language. More complicated network structures have evolved in order to deal with these scenarios. There are basically three different approaches in literature: 1.) Recurrent Neural Network (RNN) [26, 27], 2.) Convolutional Neural Network (CNN) [28] and most recently 3.) Transformer-models [29].

In this work, the Gated Recurrent Unit (GRU) [27] RNN will be used when dealing with sequential patterns. In recommendation scenarios this is typically encountered when accumulating historical clicks in a session for a user. Furthermore, it was used by many recommendation methods, e.g. [30, 31], demonstrating its effectiveness in this scenario.

The basic procedure is illustrated in Fig. 2.2. Given the input vector x_t at time t and a hidden-state of the previous time-step h_{t-1}, the GRU updates h_t through two individual gates, a reset gate r and an update gate z:

$$\begin{aligned} h_t &= (1 - z_t)\, h_{t-1} + z_t \hat{h}_t \\ \hat{h}_t &= tanh\left(W x_t + U\left(r_t \odot h_{t-1}\right)\right) \\ z_t &= \sigma\left(W_z x_t + U_z h_{t-1}\right) \\ r_t &= \sigma\left(W_r x_t + U_r h_{t-1}\right) \ . \end{aligned}$$

In this notation, all lower-case letters are vectors and the upper-case letters are weight matrices. The σ symbol actually represents the *sigmoid*-function and is not a general symbol for an activation, as introduced in the previous

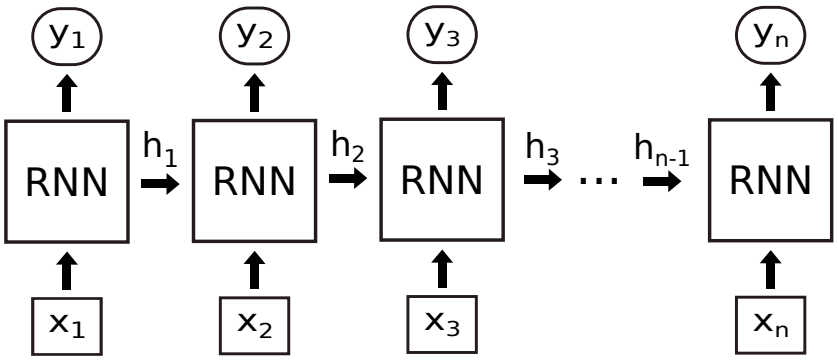


Figure 2.2: Processing of sequential data with a RNN. At each timestep, the RNN computes a latent representation h_t given the previous hidden-state h_{t-1} and the current input x_t, and returns that h_t as an output y_t. The hidden-state is usually initialized to a zero vector.

part of this section. The symbol $\odot$ represents the element-wise multiplication. In conclusion, the output of the unit at each time step is an interpolation between the last hidden-state and an approximation of the current state.

3

Related Work

The related work to this study covers three large research areas which will be outlined in the course of this chapter. At first, recommendation methods in general will be surveyed from popular static algorithms to the most recent trends. Then, current research in Reinforcement Learning will be spelled out briefly before covering its applications in recommendation in detail.

3.1 Recommendation Systems

A typical recommender system on any platform proposes a set of items to the user which he might like. How those recommendations are constructed varies a lot and has changed in recent literature. At first, static methods which are still employed in many production systems today and are typically used as baselines are introduced. Then, latest research trends of this field using Deep Learning are summarized.

3.1.1 Classic Approaches

There are mainly three different approaches for constructing recommendation sets, namely Collaborative Filtering (CF) [32], Content-based [33] and hybrid variants [34]. Most research applications for recommender systems follow the approach of CF [35]. User-based methods follow the intuition, that some users have a similar taste for the offered products and thus it is

only reasonable to pull recommendations from other users who behaved similar in the past [32]. Recommendation algorithms based on item-similarities are part of the most popular methods [36, 37]. Here, an item is represented through its co-occurrence with other items [36]. The most basic approach is called neighborhood-method and still serves as a strong baseline [38]: Given the user's last viewed item, the system recommends the product which is the closest w.r.t a distance metric via a basic k-nearest neighbor lookup. Neighborhood-based recommendation models are very popular due to their simplicity, efficiency and still accurate performance.
Another pillar of recommendation research uses Matrix Factorization (MF) techniques [39, 40] which approach CF by finding latent features that explain observed user behavior. In particular, given a matrix which stores the historical rating or interaction between users and items, this matrix is factorized into sub-matrices in order to estimate a value for items that the user did not interact with. The most famous candidates for this are probably Singular Value Decomposition (SVD) [41, 42] and Probabilistic MF [43]. Bokde, Girase, and Mukhopadhyay [44] provide a survey of different MF approaches for Collaborative Filtering.
One of the biggest flaws of the stated techniques is that they typically only involve the most recent item and can not respect temporal information. [3, 45] propose sequential recommender systems based on Markov chains for predicting the user's next action. [3] showed that already first-order Markov-chains are a good recommendation method and outperform non-temporal methods. Further, [46, 47] leveraged sequential pattern- and rule-mining in order to enhance the performance of neighborhood-based CF methods. The main problem of applying these techniques however, is that the state space grows exponentially and they thereby quickly become unfeasible in practice.

3.1.2 Neural Approaches

Deep Learning has become the most popular technique to solve non-linear problems which have been unmanageable before. Its successful applications in computer vision [48] or natural language processing [49, 50, 29] made it the de facto number one choice for any application that benefits from function approximation. Salakhutdinov, Mnih, and Hinton [51] were the first to introduce Restricted Boltzmann Machines for Collaborative Filtering to model user-item interactions, also usable in large datasets such as Netflix [52]. This model was proposed for static scenarios where no sequential interaction between the recommender system and the user was considered. As a consequence, and as GPU-powered computation became more feasible, new neural architectures which considered temporal aspects were employed

in this domain. Recurrent neural networks quickly became the tool of choice for sequential recommendation. The most popular example is GRU4Rec [31, 53] which is typically listed as a baseline for new approaches. Another example is the work of Wu et al. [54] that used Long Short-Term Memory (LSTM) cells to model temporal dynamics of users' taste. As the whole field of deep learning evolved, more architectures from pioneers such as Natural Language Processing (NLP) research were transferred to the recommendation community. This lead to the inclusion of attention [55] to RNN approaches and more efficient computation by leveraging CNNs [56]. Variational Autoencoder (VAE) [57, 58] have also found their way into the recommendation application, just as most recently Transformer-architectures [59, 60]. This list of recent neural network architectures is less than exhausted but highlights the current trend of deep learning research. In particular, new effective neural methods are developed in other domains and then adapted to the setting of recommendation. A more detailed survey of approaches can be found in [61].
However, the field of Deep Learning-based recommendation system suffers from one critical aspect - the lack of comparability. [62, 61] have shown that the evaluation procedure and choice of datasets have a high variability between papers. This limits the meaningfulness of the neural recommenders' performance if it was not evaluated in a reproducible setting. [63, 38] went one step further and evaluated recent research contributions in a common and fair setting on public datasets. They showed that the majority of neural algorithms performed worse than classic approaches, which need practically no time for initialization and inference. More specifically, the simple k-nearest neighbor (kNN) CF method introduced above outperformed nearly all of the computationally expensive neural methods.

3.2 Reinforcement Learning

Next, general algorithms for DRL and stabilization techniques that have proven to be beneficial when using neural function approximators will be introduced. Then, related work that uses DRL for the task of creating recommendations is analyzed.

3.2.1 Deep Reinforcement Learning

Sutton et al. [22] proved that policy iteration with a differentiable function does converge to a locally optimal policy. Through that, they effectively introduced REINFORCE to function approximation with neural networks.

The main motivation for the work was that value-based methods at this time failed to converge even in very simple settings. Only years later, DQN[64, 65] introduced techniques to stabilize training. The two main ideas are the usage of an experience replay buffer and target networks. The first improved data-efficiency by sampling historical interactions from a memory buffer to update the networks. The latter helps to stabilize the training procedure by using a slow moving-average version of the most recent parameters for updates. Every new algorithm that further improves value-based DRL takes advantage of these two improvements. In spite of its success, DQN was shown to suffer overestimation issues, as already discussed in the previous chapter. Since then, many improvements to overcome this issue and decrease variance have been introduced, which were combined in Rainbow [66] for a state-of-art value-based algorithm. The most essential improvement is Prioritized Experience Replay [67] which will be used in this work as well. Instead of uniform sampling from the replay buffer, the method preferably samples historical interactions where the Q-function had the highest TD-Error. This enhancement especially reduced the time until convergence and boosted overall performance significantly. Deep Deterministic Policy Gradient (DDPG) [13] was the first practical neural Actor-Critic architecture. They modeled the actor and critic with one neural network respectively and update them in an iterative manner. Here, the critic follows the typical Q-learning objective of minimizing the TD-Error, while the actor-network follows the policy gradient to maximize the critic. As of today, more methods have been proposed that solve some of the problems of these main algorithms [68, 69]. However, a thorough research of papers in the recommender domain did not yield publications that explicitly leverage these recent advances.
Another field of RL, namely Batch-RL [70] or offline-RL, focuses on the situation where an agent can not directly interact with an environment to collect experience but is restricted to a batch, i.e. a collection, of interaction data. It was shown that classic model-free algorithm performed poorly when agents merely had access to collected samples in contrast to direct online interactions [71, 72]. The authors of both publications therefore proposed adaptations of the classic online DQN for a great improvement in the offline setting. This offline setting is the starting point for every RS that wants to leverage DRL, as training is only possible from a collected dataset of recorded user trajectories.

3.2.2 Recommendation Systems with DRL

Finally, recent work which modeled the recommendation setting as a MDP and found a solution through Deep Reinforcement Learning is reviewed. A

general survey over different DRL methods for recommendation and online search was done in [73]. This section is divided into the three main RL approaches introduced in Section 2.1: Policy Gradient, Actor-Critic and Value-Function methods.

Policy Gradient Approaches. Chen et al. [74] attend the task of handling large action spaces in the recommender domain. They create a tree-based policy approach, which creates a balanced hierarchical clustering over items to reduce the full action dimension. For the construction of the tree, all items are clustered via a hierarchic clustering algorithm. Each non-leaf node in the tree is a policy-network, which yields a probability distribution over its children. Hence, starting from the root node with an observed state, the items are recommended by sequentially following the agents to the leaf nodes. The weakness of this approach is its static tree that needs to be changed as soon as new items are introduced or old ones removed. The authors chose an unconventional evaluation setting that prohibits direct comparison with other baselines.

Chen et al. [6] presented a policy approach, also based on the REINFORCE algorithm for efficient recommendation on YouTube. They focus on the setting where one can learn from historical log-data of user transitions. This means it can not be applied in a live setting where the behavior policy is learned by directly interacting with the user. Performance evaluation was merely done to emphasize the impact of their main contribution, an off-policy correction term, and later deployed directly on YouTube. No comparison with other baselines was executed.

Actor-Critic Approaches. The challenge of a huge discrete action space was also researched by Dulac-Arnold et al. [75]. Their so-called Wolpertinger algorithm, is based upon the Actor-Critic Framework where an actor directly proposes an action in the item feature space. Given this proto-item, an approximated kNN lookup selects the closest M neighbors from the item set and recommends the top-k items as a list that have the highest action value. The framework is trained with the DDPG [13] algorithm, which alternately optimizes the actor's policy and the critic's value estimation. Their evaluation in a simulated recommendation setting however, only showed that the method was able to learn a policy that could increase its earned reward per epoch over time. There was no direct comparison to any baseline and specifics of the simulation environment remained unknown. One inconvenience of this approach is the involved kNN method because fast lookups are only approximated and can not guarantee that the real neighbors are

returned [76]. However, it is necessary for the transition from continuous to discrete action space and feasible in practice.

This principle of calculating a representation in feature-space which will then be mapped to a real item via kNN was adapted by future works for RS where multiple items are recommended at each time-step. Basically all following actor-critic methods involve the kNN at some point.
Sunehag et al. [77], also involved in [75], formally defined a Slate-MDP, i.e. each action consists of a set of items. By doing that, the agent starts to combine single items in a more coherent way than by simply choosing the top-k elements. Zhao et al. [7] proposed another approach to recommend a list of items. Here, the actor proposes a vector of size $k \cdot d$, where k is the length of the recommendation list and d is the feature dimension. Each vector slice of length d is then mapped to its nearest neighbor and the critic assesses the whole list with its expected reward. Zhao et al. [78] take it one step further and create a recommendation agent which jointly adapts to user's real-time feedback and focuses on a proper display alignment as a 2D-page of the recommended actions. Therefore, the actor estimates an action, which again is a list of feature representations, by calculating a deconvolution function [79] from the latent representation. Again, this list of proto-vectors is mapped to actual items via kNN. The authors further limited the candidate items to a subset through adoption of an item recalling mechanism which picks similar items for each item the user interacted with recently. Every time a user clicks on the next item, a number of similar items are introduced into this selection set. Performance was compared to the static but strong baselines from Section 3.1.1, DQN [65] and Wolpertinger [75], showing superior performance. However, the used dataset from a real e-commerce company was not specified further, i.e. it is impossible to validate those results.
The dynamic environment in recommender systems was also studied by Liu et al. [80]. They aggregated the current state by an average pooling layer over the last interacted items. In contrast to previous listed methods which leveraged some kind of recurrent structure, this setting can not capture the sequential property of the current session. The authors defined an action to be a ranking function, represented by a continuous parameter vector. The item to be recommended is picked through the dot product between the ranking function a and each item in a candidate subset.

Zhao et al. [81] created a recommendation framework for different scenarios, i.e. one agent specialized e.g. on an overview page and one on item-detail-pages, which are all trained jointly. The state transition probabilities are explicitly modeled for the given scenario, in contrast to previous model-free algorithms. Compared to the online setting, where the user immediately

reacts to the recommendations of a policy, the ground truth is fixed in the recorded log data. This means, whatever the current agent's policy predicts, it has no real connection to the ground truth, as this prediction might not have been available at this moment, and thus the reaction of the user is unclear. This is the classic case of Batch-RL as introduced above. In order to compensate for this gap, they deduced that the action output of an actor should be as close as possible to the critic's input action. This was accomplished by minimizing an auxiliary regression-loss function between the real and the proto-item.

Q-value Approaches. Chen et al. [82] focused on decreasing the high-variance and biased estimation of the reward due to the dynamic environment. The first challenge was addressed by leveraging a stratified random sampling technique when drawing from the widely used experience replay method [65] for Q-learning approaches. To mitigate the bias introduced by the environment, e.g. that some users' behavior depends more on the time of day and has nothing to do with the proposed recommendation algorithm, they adapted the reward function to be the difference between the current immediate reward and the mean of a handful of benchmark users. The proposed method was exclusively evaluated by online testing on the Taobao platform.

While the majority only considers positive feedback in form of a rating or item-interactions, [83] incorporated negative feedback on interactions into the recommendation application. This results in a Q-function approximation with input (s_+, s_-, a), with an extra pairwise regularization term. This work can be seen as more of a re-ranking approach, where an underlying fixed policy π creates recommendations, which will be ranked according to their respective Q-values. This method can not be used for a comparative evaluation as negative feedback information is usually not available.

Zheng et al. [84] propose a method for news recommendation with a dueling network architecture [85]. Users and items are represented by vectors constructed via extensive feature engineering. They further took the activeness of the user into consideration. Concretely, if a user returned to the application after leaving a given session, the reward for the last recommendations is higher. Furthermore, another exploration strategy than the typically applied ϵ-greedy is proposed. Experiments only weakly supported the efficiency of this exploration strategy, especially in the offline setting.

SlateQ [86] is another form of slate prediction which decomposes the Q-value of a slate into a combination of the Q-values of its respective items. Therefore, it is assumed that if the user consumed a single item from a slate,

the reward which was returned by the environment for executing slate A only depends on the user's reaction to that particular item rather than the whole list. Experiments were executed on an artificial user simulator and on the YouTube live system without comparison to any other baseline (besides random). The proposed model is heavily based on a user choice model $P(i|s, A)$, i.e. the probability for clicking item i when observing state s and a slate of actions A. However, in public datasets, only implicit feedback is available rather than the information how a user reacted to a full recommendation list of items.

4

Method

In this chapter, the methodology of this work is illustrated. At first, the task for the evaluation will be defined formally. Next, practical challenges of applying Reinforcement Learning to the recommendation setting are highlighted. Then, the model selection process is explained, together with a summary of each model's functionality. At last, the exact evaluation procedure is outlined for the given setting.

4.1 Task Definition

In this work, the interaction between a user and the recommender system is modeled as a Markov Decision Process, as generally introduced in Section 2.1.1. The exact interaction in the recommendation setting is illustrated in Fig. 4.1. Here, the system represents the agent which recommends a list of items to the user in order to receive the highest possible cumulative reward. The user personifies the environment who observes these recommendations and returns implicit feedback for those predictions. Implicit-feedback means, that the agent does not receive an explicit reaction, e.g. a rating, that the user liked the recommendation, but it is implied that he does when he interacts with one of the products in it. In terms of recommender systems, the evaluation task in this work is the next-item prediction.

In particular, the components in the quintuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ which defines the MDP, are defined as follows:

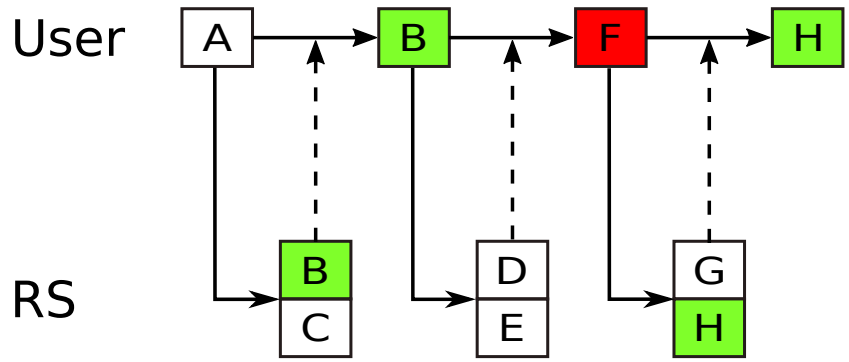


Figure 4.1: Example of a typical interaction-based recommendation scenario. The user first interacts with item A. Based on this observation, the RS creates a recommendation list that contains suitable next items. The user consumes this list and provides implicit feedback by clicking on a recommended item *or* selecting a different one. The agent gets a positive reward to reinforce the current policy only if one of its recommended items was consumed.

- **State**: A state is represented by the last m items that the user interacted with. For the first states in a trajectory, the length is shorter than m. The initial state s_0 in a trajectory is given by the first item, i.e. the agent only provides recommendations after observing at least one item interaction.

- **Action**: An action $a \in \mathbb{R}^k$ is a list of k items which are recommended to the user. Each item has a corresponding vector representation $i \in \mathbb{R}^d$ that is learned along the process or computed beforehand.

- **Transitions**: A state transition is one step in the user's click trajectory. The transition probability reflects the user's interest in consuming an item from action a, given the historical items in state s. The transition function is given by or created from a dataset of recorded user trajectories.

- **Reward**: The recommendation system predicts a list of items, which the user might be interested in and receives a reward according to the user's feedback. In this work, the reward is binary, i.e. if the user clicks on *any* item in the recommendation list, the reward is 1 and else 0. This corresponds to the setting in [86], where a reward depends exclusively on the clicked item in contrast to the whole list.

- **Discount factor γ**: The future reward discount is set to a fixed con-

stant, typically close to 1. This means an agent aim to learn a long-term strategy in contrast to a myopic one.

Subsequently, one episode of Reinforcement Learning is defined as a fixed-length trajectory $\tau = (s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T)$, representing the interaction between user and agent, starting from an initial state s_0. Here, r_{t+1} is the immediate reward for executing action a_t in state s_t, received at the next timestep. The set of available items is notated as $\mathcal{I}$ and $i \in \mathcal{I}$ is one item, e.g. a movie. An action $a \in \mathcal{A}$ is a list of k items where a^j is the item at position j. The superscript notation is used here because it is typical to use a_t to refer to the timestep. The cardinality of the full action space itself is thus exponential on the number of items.

4.2 Challenges

The application of Reinforcement Learning to recommendation systems is not as straight-forward as one might think. Dulac-Arnold, Mankowitz, and Hester [87] highlighted the main challenges which need to be taken into account to bring a RL-method to production.

One of the main challenges in training a new recommendation method is that it needs to be done offline on collected data. Static or supervised machine learning approaches can directly optimize on that data without special attention. RL on the other hand, especially on-policy approaches, need to compensate for the distribution mismatch through alignment techniques such as Importance Sampling (IS) [88]. Furthermore, one needs to ensure that a new recommendation model is working at least as good as the previous one before applying it online. This however brings special focus to the evaluation method. As the log-data only shows the user's behavior under a previous recommender policy, one does not know how a user would have reacted to different recommendations. The typical procedure in this field [78, 75] is to create a user simulator, based on the dataset and evaluate the agent based on its performance there. The creation of a good user model however is just as hard as training the recommender system itself, and is further subject to modeling errors [89]. For this reason one commonly finds various simulator choices [78, 75, 90] which further restricts comparability.

Another challenge is the large action and state space, as the number of items in e-commerce stores or media platforms easily exceeds the millions. Most general research in RL only considers settings with only a few discrete actions, e.g. Atari games. As outlined in Section 2.1, expected state values only converge to the true value if a state was visited sufficiently often. Furthermore, the best strategy can only be found if many actions have been tried

in any state. The challenge of the large state space can be contained through function approximation. In particular, by finding similarities in states, an agent can generalize better over the action selection process. A huge action space remains an open problem and some methods which target this challenge are introduced in the next section.
Finally, a platform typically removes items and adds new ones on a daily basis. This demands scalability of the used methods to update the latest model quickly.

4.3 Method Selection

In the following, details of the method selection process are illustrated, followed by a summary of the chosen algorithms. Then, two additional methods are taken into considerations that have been proven beneficial in other domains and have not been demonstrated in the RS setting before.

4.3.1 Recommender System Publications

Taking into account methods that proposed general algorithms and evaluated them on a recommender setting, twelve candidate approaches were found (excluding exclusive arXiv.org pre-prints). An extensive list was already presented in the related work (Chapter 3) of this work. Unfortunately, only one [74] of those candidates provided their code on an open-source platform and even this one was not usable due to the lack of a modular structure to use a custom simulator in a straight-forward way. For this reason, the focus was on those works which where descriptive enough to be implemented as accurate as possible. Some algorithms, e.g. [78], which focused on presenting the items in a certain layout, had to be left out because the setting was too special. In the following, a description of each selected algorithm and its main contribution is illustrated. Some factors had to be excluded by some methods as the evaluation setting in this work differed from the original one or the leveraged information is not available in public datasets. Fortunately, at least one candidate from each of the three main RL method families is included. The approaches are presented from Policy Gradient over Actor-Critic to Q-Learning. The two Policy Gradient approaches are *on-policy*, all others are *off-policy* algorithms. Each method, except LIRD, is updated w.r.t to the the one item of a list that was responsible for getting a reward. If none of the recommendation items is selected by the user, it is simply the first item of the list.

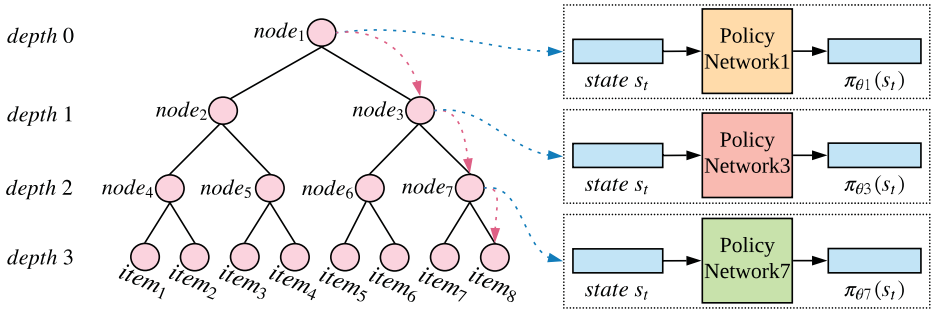


Figure 4.2: Basic architecture of the TPGR approach. There exists one separate policy network for each inner node which computes a probability distribution over its children. Hence, suitable recommendations are created by successively following the policies from the root node to the item ids in the leaf nodes. [74]

Tree-Based Policy Gradient (TPGR). The first candidate is [74] who proposed a tree-based policy gradient approach. With their work they published one solution to handle the challenge of large action spaces in the recommender setting. Therefore, they constructed a hierarchical clustering of all items in order to form a balanced tree. They compared different item representations and found a rating-based Principal Component Analysis (PCA) approach to be the most effective for clustering. Given a user-item-interaction matrix, an item representation is the column of this matrix, i.e. the interaction behavior of all users with this item. This sparse vector is then reduced to a dense latent representation through PCA.

After this hierarchical clustering step, they obtain a balanced n-ary tree where all items are leaves, as shown in Fig. 4.2. Each non-leaf node in the tree is a policy-network, which yields a probability distribution over its children. Hence, starting from the root node with an observed state, the items are recommended by sequentially following the policies to the leaf nodes. This algorithm is updated with the REINFORCE framework, utilizing a Q-value baseline for enhancement. This baseline is effectively the discounted reward-to-go version, introduced in Section 2.1.5. The reward function that involves consecutive rating information was not included because this is a technical information extracted from the used datasets. The original tree construction routine from their GitHub repository[1] was used and the depth of the constructed tree was set to $d = 2$ as used in the paper.

Recommender systems typically learn from data that was produced by following another policy β. In order to apply an on-policy method in an

[1] `https://github.com/chenhaokun/TPGR`

offline setting, one has to introduce a correction factor for the fact that the recent policy gradient estimates are not created by the current policy [87]. One possibility is to leverage IS to compensate for this distribution mismatch:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\beta}\left[\sum_{t=0}^{T} \frac{\pi_\theta(\mathbf{a_t}|\mathbf{s_t})}{\beta(\mathbf{a_t}|\mathbf{s_t})} \nabla_\theta \log \pi_\theta(a_t \mid s_t) Q^{\pi_\theta}(s_t, a_t)\right] . \tag{4.1}$$

The main difference to the original REINFORCE with baseline (Eq. (2.20)) is that the expectation is over the behavior policy β which is then compensated for by the IS likelihood ratio $\frac{\pi_\theta(a_t|s_t)}{\beta(a_t|s_t)}$. Additionally, simple weight clipping of this term was applied to restrict the otherwise unbounded variance of the estimator. In Eq. (4.1), a_t is one item to be consistent with the notation in the preliminaries chapter.

As publicly available datasets do not come with the RS-model that lead to the collected trajectories, the logging policy β must be approximated. Here, it is modeled as an additional MLP and is optimized jointly through a cross-entropy loss. This is a good approximation if the old policy was a myopic RS that was optimized for the immediate reward, i.e. practically any method that did not model the setting as a MDP.

Top-K Off-Policy Correction (CORRECTION). The off-policy compensation for on-policy RL methods was further studied by Chen et al. [6]. They proposed an additional term in the optimization that includes a correction for drawing a top-k list as one action. Concretely, they multiply the term under the sum of Eq. (4.1) by $\lambda_k(s_t \mid a_t) = k(1 - \pi_\theta(a_t \mid s_t))^{k-1}$:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\beta}\left[\sum_{t=0}^{T} \frac{\pi_\theta(a_t|s_t)}{\beta(a_t|s_t)} \lambda_{\mathbf{k}}(\mathbf{s_t} \mid \mathbf{a_t}) \nabla_\theta \log \pi_\theta(a_t \mid s_t) Q^{\pi_\theta}(s_t, a_t)\right] .$$

This additional factor resembles the probability that *item* a_t is drawn from the categorical distribution π_θ without replacement and appears in the final list of size k. They used a MLP with two heads, where one represents the target policy π_θ and another head approximates the behavior policy β that created the logs. In the original work, the authors also add softmax approximation methods to deal with the large action space on the YouTube ecosystem. In this work, the normal softmax distribution is used as the action spaces of selected datasets are smaller.

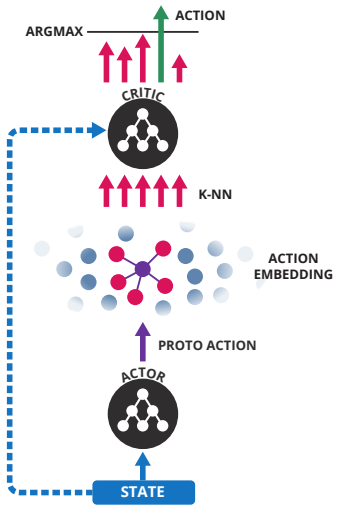


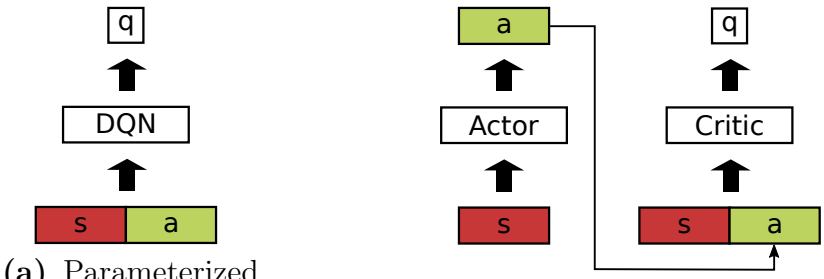
Figure 4.3: Summary of the Wolpertinger algorithm. An actor computes a vector in embedding space which is mapped to a subset of real items through a kNN lookup. Those closest items are ranked by a critic to select the k items with the highest expected return. In this illustration, an *action* is it be understood as an item. [75]

Wolpertinger. The next candidate is the Wolpertinger algorithm [75] which is illustrated in Fig. 4.3. The actor and critic of this framework are represented by one neural network each. First, the actor computes a proto-item in the item feature-space. Next, an (approximate) kNN lookup method returns the closest n items according to the euclidean distance. The feature representation of each item in the subset is then concatenated with the state vector and valued by the critic. Then, the top-k items that have got the highest Q-value are chosen as the recommendation for the user. The authors found that setting $n = 0.1N$, where $N = |\mathcal{I}|$, achieves similar performance as $n = N$. This means, that merely 10% of all available items need to be evaluated by the Critic which is a large advantage to Q-Learning approaches. As an action in this work is defined as a list of items, the top-k items with the highest action values are chosen.
This approach follows the DDPG algorithm but needs to make an adaption because the neighbor lookup function is not differentiable. In particular, there is a gap between the proto-item that was computed by the actor and the action that was evaluated by the critic. For this reason, the authors chose to update the actor via Policy Gradient w.r.t to the computed proto-item. The critic on the other hand makes an update step through the typical TD-Error using the real representation selected from the kNN lookup.

The main contribution of this paper focused on the challenge of large action spaces. It is also used as a baseline by many other papers in this field, including some of those selected here. The kNN lookup is computed by a matrix multiplication instead of an approximation. This is feasible in this work as the number of items are still of a manageable size in the available datasets and Graphics Processing Units (GPUs) are highly optimized for this. Furthermore, the version that sub-samples the full item set to $n = 0.1N$ is implemented.

List-Wise Recommendations (LIRD). Zhao et al. [7] explicitly extend the Wolpertinger algorithm to a list-wise recommendation scenario. In fact, there are only two distinctions between the algorithms. First, the output of the actor is of size $k * d$ where k is the size of the recommendation list and d is the dimension of items' feature space. The same holds for the action input of the critic. The architecture of Actor-Critic is additionally sketched in Fig. 4.4b. Second, the computed vector represents a list of ranking functions in contrast to a list of proto-items. The actual presented item list is then created by computing the dot product between each ranking function of size d and all available items in the item set. The highest scoring item is then selected for each slot in the list. In theory, this allows this method to recommend more diverse actions in contrast to Wolpertinger' top-k approach that yields similar items.

News Recommendation (NEWS). Finally, [84] is picked as a candidate for Q-learning. They used a DQN in the same form as the critic of Actor-Critic methods, i.e. the input is a concatenation of state and item embedding and the output is a scalar - the action value. The advantage of this architecture style is that in large action spaces, the network can generalize over different actions through similarities in their item representation. In order to include exploration on a live system, that does not annoy the user with purely random choices given by ϵ-greedy strategies, they proposed a Dueling Bandit Gradient Descent exploration technique. However, experiments only weakly supported the gains of this adaptation. In addition, the modeling of User Activeness is neglected as this information is not given in publicly available data-sets. In conclusion, the model represents a typical parametric DQN as shown in Fig. 4.4a.



(a) Parameterized DQN

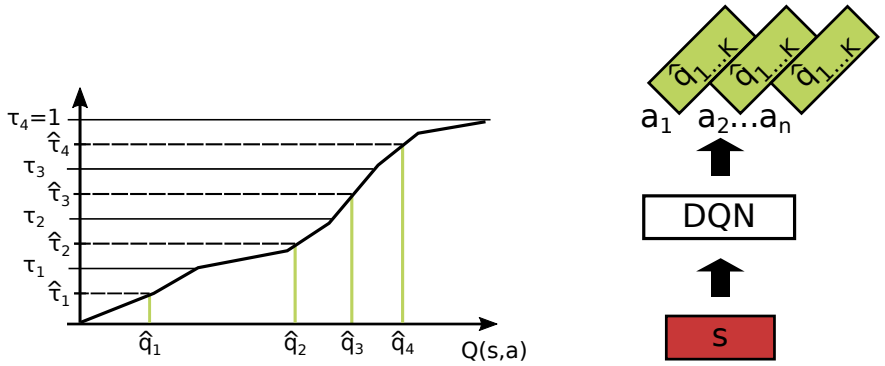
(b) Actor-Critic

Figure 4.4: The parameterized DQN architecture has exactly one output value that resembles the Q-value of the state-action pair of the input. This network also functions as the critic in Actor-Critic approaches. Here, action a is the feature representation of one item, except for LIRD that explicitly works on the whole list.

4.3.2 Additional Approaches

In the previous subsection,several algorithms were introduced and selected for this evaluation. However, the off-policy approaches use model-free algorithms, in particular Double-DQN or DDPG, that have been shown in other domains to fail in the offline batch setting [71]. For this reason, two additional, and more recent, methods were chosen that are supposed to enhance the performance. Both methods only introduce slight adaptations to the DQN algorithm.

QR-DQN. Quantile Regression DQN (QR-DQN) [91] is a distributional reinforcement learning method, one of the most recent trends in DRL. Classic Q-Learning aims to learn the expected discounted return from taking an action a in state s. In contrast, distributional RL estimates the full return distribution by predicting quantiles that are fixed to certain probability thresholds. Quantiles are cut points that divide a probability distribution into intervals that contain the same probability.
Therefore, the expected return $x = Q(s, a)$ is a random variable, drawn from an unknown distribution Z. Furthermore, the number of quantiles into which the whole return space is separated is a hyperparameter K. For example, let Fig. 4.5a be the Cumulative Distribution Function (CDF) of Z for the current state and chosen action. Additionally, τ_i are fixed probability intervals of equal size and $\hat{\tau}_i$ is its respective center. In the example, the distribution was separated into $K = 4$ quantiles. The estimates $\hat{q}_i$ can be interpreted as percentiles, e.g. with a chance of $P(X \geq \hat{\tau}_1) = 1 - P(X < \hat{\tau}_1) = 1 - \frac{1}{4} \cdot \frac{1}{2} = 0.875$,



(a) CDF of return distribution Z for one item

(b) QR-DQN

Figure 4.5: Illustration of a CDF with quantiles and the QR-DQN network architecture. The network estimates the mean action-value $\hat{q}_i$ of each quantile $\hat{\tau}_i$ of the return distribution.

the policy expects a reward of at least $\hat{q}_1$. The limits of $\hat{q}_i$, i.e. the minimal and maximal possible cumulative reward, are typically given by the environment. It is the goal of the model to estimate the mid-point return $\hat{q}_i$ for each quantile. This is achieved by minimizing a distributional Huber-loss in the TD-Error. In this work, QR-DQN is leveraged to demonstrate the effectiveness of more complex advances in RL. Therefore, the interested reader is referred to the original publication for a detailed theoretical description of this procedure.

In practice, the model is realized with a DQN that does not only output one value, the expected return, but a K-dimensional vector, as displayed in Fig. 4.5b. This has been shown to be superior to the plain estimation of the return expectation, i.e. classic value methods.

REM. Random Ensemble Mixture (REM) [71] is a much simpler approach but has shown comparative success to QR-DQN in offline RL settings. It is mainly inspired of performance gains shown in supervised learning from using ensembles of models instead of merely a single one. In particular, when using the parametric DQN architecture from Fig. 4.4a, one not only computes one Q value for the given state-action pair but K. These value estimations are then accumulated by a weighted average. The K weights for this are drawn from the uniform distribution and then scaled to form a probability distribution.

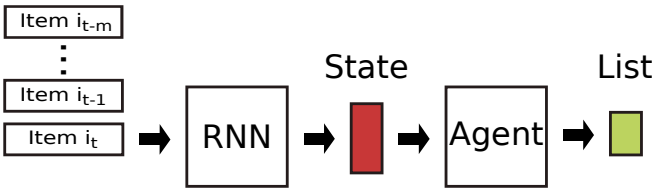


Figure 4.6: Overview of the evaluation framework for all methods.

An action is selected by choosing the items that have the highest *mean* value over all heads. While it was also possible to use the same architecture of QR-DQN, the parametric version is used as it was the original choice in the paper. In addition, the double Q-Learning (see Eq. (2.14)) version of DQN is implemented for enhanced stability [92].

4.4 Evaluation Procedure

Next, the exact way of proceeding for a comparative evaluation of the previously mentioned methods is described. For a fair setting, a fixed framework around each agent is constructed so that none of the methods has got an information advantage. Therefore, an evaluation framework is built that uses the same state module and agents can be exchanged through a plug-in architecture, as illustrated in Fig. 4.6. The reason for a shared state module is the inconsistency of this part in the approaches. In particular, Wolpertinger did not consider multiple previous items as one state but only the very last one. Others, CORRECTION and TPGR, used RNN cells that have less complexity than the popular GRU [27] version and are hence merely faster. As the GRU-RNN is already a popular choice in supervised learning [31], it will be used for all approaches in this evaluation. Following [7], it is reasonable to limit the number of items in a state to maximal m items, especially for computational efficiency.

The best evaluation for any recommendation method would be to deploy a model, that was trained on offline logs, and monitor its behavior via A/B-tests. However, this is practically never a real option [93] due to three reasons. First, some systems simply prohibit the straight-forward usage of different versions of a recommender system by design. Second, a model needs to be ensured to be at least as good as its predecessor, otherwise the company might loose profit [87]. And finally, researchers from the public domain are limited to open-source datasets and have no access to deploying their method after all. Further, [94] studied the impact of exploration for recommendation algorithms and found a large reduction in user experience when too many

random or unrelated items are recommended.
In this work, three different evaluation settings are executed. In the following, each scenario is elaborated in detail.

4.4.1 Pure RL on Log-Data

The first evaluation setting resembles a naive approach to directly apply the respective Reinforcement Learning routines to the offline data. This is the same strategy in which a RS with supervised learning would be trained and is the most likely idea when coming from this field. This setting was executed first because some publications describe their evaluation procedure in such a superficial manner that it was expected that they followed this approach.

Setup. Each agent follows its optimization procedure on the training dataset as introduced in the previous subsection. This means, given the current state s, the agent computes a recommendation a which then leads to a reward $r = 1$ only if the very next item in the collected offline data is in a. However, finding the correct items for a heavily relies on getting a reward at some point for an item through exploration. Hence, the bigger the item space, the less likely it is that an exploration strategy, such as ϵ-greedy, will ever receive a reward.

For this reason, only the smallest available dataset in terms of number of users and item space is considered for this first evaluation. In addition, the exploration process is guided by providing agents a candidate set of $k = 500$ items, from which one of them is the target of the current state transition, to speed-up the learning process as in [75]. For a dataset with a large item space, this restriction would also be helpful, nevertheless each individual item would be covered very rarely which is inconvenient especially for value-functions.

After a fixed number of episodes all agents are tested with a greedy behavior policy on the previously unseen test trajectories.

Pretraining. Further, the impact of different levels of pretraining is studied. Concretely, each approach is trained first completely from scratch, then with Word2Vec [49] representations for each item and finally in a full supervised setting. Information about the Word2Vec embeddings is given in the subsequent chapter and the full pretraining strategy is outlined in the following.

Each agent is trained for a fixed number of episodes n on the trajectory data and the best model for the RL procedure is chosen by early stopping on a validation set. After pretraining, the RL procedure starts again from

the first trajectory in the dataset.
Given a trajectory $\tau = [i_1, i_2, i_3, \dots]$, each model is optimized towards predicting the next item correctly. For example, given the state i_1, i_2, the target to be predicted is i_3.
TPGR is trained using a typical cross-entropy loss that increases the likelihood to predict the target item in the current state.
Q-network methods minimize the mean squared error (MSE) between the computed action-values and a one-hot vector. The target vector is 1 at the position of next item id and otherwise 0. This increases the expected value for the next item while keeping all other expectations close to 0.
A sample version is used for parametric Q-networks, concretely NEWS and the critic in Wolpertinger and LIRD. Therefore, 20% of the item-space was sampled as zero-value candidates to the correct target item. This is done purely for efficiency reasons as one had to compute a value for all items in $\mathcal{I}$ otherwise.
Actor-Critic methods have one loss-function for the actor and critic respectively: 1.) The actor minimizes the MSE between its proto-item in the output and the target's Word2Vec representation. In case of LIRD, the distance for the first slot in the output list is minimized, while the rest is not considered. 2.) The critic, a parameterized Q-network, follows the one-hot vector MSE objective as described above.

The goal of any training procedure is to learn as much as possible from the provided training trajectories. In this setting however, only a fraction of real state transitions have been observed by agents as they are subject to discovery through exploration. Consequently, agents can not leverage the recorded user trajectories fully to develop a good policy and are hence practically incomparable to other non-RL approaches that did have access to every state transition in the data.

4.4.2 Offline Evaluation

In order to fully leverage available data, the paradigm of Batch-RL [70] is followed where the methods are fully guided on the training set. This is the method of choice when only log data is available and no further interaction between the agent and environment is possible. As in the previous supervised setting, each family of RL methods is optimized by minimizing different loss functions.

Q-networks, as in value-based methods and the critic in Actor-Critic, are typically updated by the TD-Error (see Section 2.1.4) with samples from an experience replay buffer. In this setting, agents consecutively fill the buffer with the logged transitions in the training set and sample from those

rewarding experiences.

Policy Gradient methods have to update two models: 1.) the target policy and an approximation of the behavior policy that created the logs as an off-policy correction. The behavior model is trained to optimize a myopic strategy through a one-step cross-entropy loss. This is identical to the supervised update in the previous evaluation setting. The target policy is optimized with the REINFORCE algorithm as outlined in the previous section.

For the Actor-Critic methods, one updates the actor with a MSE to the target item's representation. Fixed Word2Vec item representations are used for *all* models because the Actor-Critic methods otherwise collapsed in the straight-forward batch setting. If the embeddings were trained jointly, the most simple and optimal solution would result in an identical embedding for all items, e.g. the zero vector. Elaborating counter measures for this problem is out of scope for this work, as there is the possibility to circumvent it with pretrained embeddings for all approaches.

After training on the offline data, all methods are tested on a test dataset with previously unseen users. This reflects the moment where one wants to ensure that the models have learned a good general policy and there is no risk to deploy them.

4.4.3 Online Evaluation

Offline evaluation can offer a glimpse of a model's capabilities but it is still restricted to the behavior policy recorded in the log data. In particular, there is a gap between the agent's recommendation and the user feedback. A user might like a product in the recommendation list but as it was not recorded in the data, the agent receives a zero reward.

As A/B-testing is not available, one follows literature [78, 7, 95] and creates a simulator based on the provided offline data to evaluate the agents to at least partially overcome this gap. A well modeled simulator can interpolate a feedback for an item in a user trajectory that was not recorded in the dataset. This allows an agent to actually get feedback for recommendations that have not been presented during recording of the log data. In addition, this leads to a real interaction between the user and the RS. In the previous batch setting, the state transition was independent of an agent's recommendation. Concretely, no matter what the agent recommended, the next item in the state was obtained from the recorded user trajectory.

In an online simulation, however, the user directly responds to a recommendation. This means the trajectory that a user follows is directly influenced by the recommendation engine, as it is found in practice.

Unfortunately, the manner of how this simulation environment is used, drastically differs between each publication. For example, [7] trained the agent only on the offline batch and used a test-set user simulator as a second evaluation. [78] on the other hand, trained the agent on one simulator and evaluated it on another one, which was created from user trajectories not involved in the training environment. Further, [95] more or less mixed the previous two approaches. In particular, an offline batch is created by a myopic policy on the simulator and this batch is then used for training the RL agent. Later, the agent is evaluated on the same simulator from which the data was collected originally.

As one can see, there is no agreement on the (online) evaluation scenario for RL-based recommender systems. In this work, the test-only approach from [7] is followed for two reasons: 1.) When an agent learns in a simulated environment, the simulator itself needs to be ensured to realistically mimic user behavior. Already small modeling errors could be exploited by a policy to perform well in the simulator but the corresponding policy would be sub-optimal. As there exists no common simulator from the research community, care must be taken when choosing a user behavior model. 2.) Training every method additionally to the offline setting on the simulator takes a lot of resources. As multiple random seeds must be used for each method and dataset respectively, the total number of computation jobs would be extremely high.

Therefore, each trained agent from the previous setting is evaluated in a simulated environment that was created from the test dataset. In particular, the one random seed parameterization that achieved the highest performance on a validation set during batch training is used. This is the equivalent of a real scenario where one would deploy the most effective offline model only. The RS is then executed on the test set with a greedy policy, i.e. without exploration. Each approach is ran for multiple random seeds because the simulator itself contains stochastic components and using only one run could be merely subject to luck.

Additionally, items that appear in the current state are masked so that they can not be recommended again. Without the masking, each RS recommended the same item repeatedly which is undesirable.

5

Evaluation

This chapter contains the evaluation of the chosen methods for the introduced RL task in a shared setting. Therefore, the publicly available datasets which are used for the experiments are introduced. Next, the simulator algorithm for online simulation is described. After that, the metrics for evaluation are presented, followed by baseline methods. Finally, the results of the proposed evaluation settings are presented and interpreted in chronological order.

5.1 Datasets

The following three popular recommendation datasets will be used which are publicly available:

- **MovieLens-1M**[1] **[96]:** The set consists of one million movie ratings from users. This 1M-subset contains ratings from 4/2000 to 2/2003.
- **MovieLens-25M:** This version contains twenty-five million movie ratings, over the full time period from 1/1995 to 9/2019.
- **Taobao**[2]**:** Taobao is one of the biggest e-commerce platforms in China. This sampled subset contains user trajectories from around 1 million

[1] `https://grouplens.org/datasets/movielens/`
[2] `https://tianchi.aliyun.com/datalab/dataSet.html?dataId=649`

users in a period of one week in 2017. The dataset tracks different kinds of user interactions such as click, add-to-cart or buy events.

This selection was chosen due to a number of reasons. First, MovieLens is one of the most famous recommendation datasets available and can be found in related work. Even though it is not ideal for the next-item prediction task [96], it usually works well in practice [97]. The data from Taobao consists of many user trajectories from a real e-commerce store. As this records session data, it is well suited for the task of interactive recommendation.

Preprocessing

All given datasets are preprocessed by a handful of steps. In particular, a minimal item support of i_{min} is enforced and then all users which have less then t_{min} or more than t_{max} clicks in their trajectory are removed. As there are some differences for MovieLens (ML) and Taobao, they are treated individually in the following.

MovieLens. First, all rating values are set to 1 as it is interpreted as implicit feedback in this work. In contrast to other work [74], all ratings in the dataset are considered as feedback, instead of only those with a positive rating higher than 3. The intuition is that even if a user gave a movie a bad rating, he still thought he would like that movie before watching it, otherwise he would not have watched it in the first place. In order to eventually be able to take advantage of the provided content information for ML-25, all users were removed that interacted with an item with missing content data. Luckily, this only concerned a tiny fraction of the dataset. Finally, the minimum item support was set to $i_{min} = 5$. The original item distribution is displayed as a log-log plot in Fig. 5.1. As visible, this choice did not concern the ML-1M variant much. The full item space of ML-25M, however, was decreased heavily. As not all methods compared in this work are specialized for large item sets, it is crucial to limit the total amount to a size which is manageable by all approaches. User trajectories included in the MovieLens datasets have already got a minimum length of $t_{min} = 20$. An upper boundary at $t_{max} = 300$ is set as illustrated in Fig. 5.2 as these trajectories mostly stretch over a long time and therefore might include a drift of interest. As the figure indicates, only few trajectories are removed through this step.

Taobao. There are three consecutive steps for preprocessing the Taobao dataset. 1.) As the original set is huge and has a full item-space with more than 4M items, at first only user sessions are selected that contain

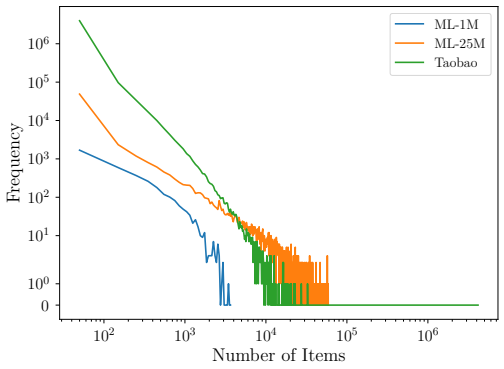


Figure 5.1: Original Item distribution for each dataset as log-log-graph.

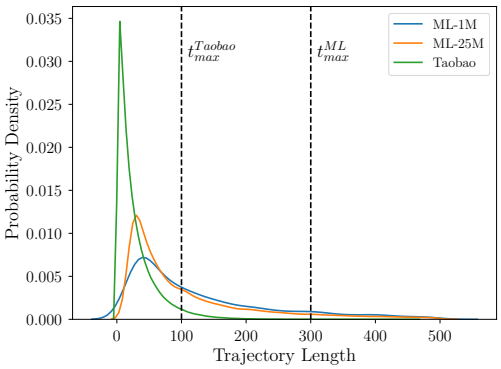


Figure 5.2: Probability density of the session length distribution. The graph of Taobao shows the distribution after the second preprocessing step.

the most popular 25k items. This dimension was chosen to contain the not manageable item space for some approaches under consideration as described above. Additionally, this implicitly sets a high item minimal support of $i_{min} = 527$. As indicated by Fig. 5.1, the original distribution had a very long tail, where the majority of items only occurred once in the dataset.
Next, all items which are not in the top 25k most popular ones are removed from the selected sessions. While this changes the actual interaction path of the user, it still resembles the user's interest path. This is especially a valid assumption, as the dataset consists of session interactions and thus the click path follows the same intrinsic goal. 2.) The removal of items which are not in the top 25k, leads to a consecutive repetition of the same item in many cases. As these transitions are useless in terms of recommendation, successive items are merged. For example, an intermediate click-trajectory of $(1, 2, 2, 2, 3, 2)$ is thereby transformed to $(1, 2, 3, 2)$. The *resulting* session length distribution is shown in Fig. 5.2. 3.) Then, the user session lengths are restricted to be between $t_{min} = 5$ and $t_{max} = 100$. An upper bound of the session length is applied due to the assumption that those users are either robots or browsing without a goal because the original session must have been much longer.
After all these steps, there were around 778k user sessions remaining that fulfilled the constraints from originally one million users. From all of these trajectories, 60k were randomly sampled to speed-up the evaluation process. This subsampling is also common practice for another large dataset in the CF domain, Netflix [52], in order to decrease the set to a reasonable evaluation

	Dataset	#Users	#Items	#Interactions	Density
Original	ML-1M	6,040	3,706	1,000,209	4.5%
	ML-25M	162,541	59,047	25,000,095	0.3%
	Taobao	987,994	4,162,024	100,150,807	0.002%
Preprocessed	ML-1M	5,080	3,403	497,938	2.9%
	ML-25M	129,078	13,461	10,303,011	0.6%
	Taobao	60,000	25,000	1,691,398	0.1%

Table 5.1: Statistic information for all datasets before and after preprocessing.

size [98].

Finally, the item ids are rearranged in descending order based on their frequency. Hence, item 1 is the most popular and item n is one of the niche items in the long tail. This is mainly done in order to investigate the behavior of agents in the evaluation. Table 5.1 summarizes statistics for all datasets before and after preprocessing. Through the applied steps, the density of Taobao and ML-25M was raised significantly which is a good property for machine learning applications.

At the end, each dataset is split into three parts chronologically after sorting the user trajectories by the provided timestamp. First, the last 20% of users create the test set. The remaining users are split into a training and a validation set with a ratio of 70:30. The latter set will be used for selecting hyperparameters and measuring the generalization for early-stopping model checkpoints.

Item Representations

This work leverages Word2Vec [49] vectors as item embeddings because the provided content information for all datasets was not meaningful enough. Concretely, for ML-1M and Taobao, merely categories are provided which, however, would result in many identical representations.
The vectors are created using the popular Gensim [99] package and are trained exclusively on the training set. The context window size was set to 3 and it was trained for 10,000 iterations, when convergence was visible on the loss graph. The rest of the parameters stay the default values of the framework. Vectors of size 16, 32 and 64 are created but no significant benefit was visible from using a dimension larger than 16.
In order to close the gap between unseen items in the test and train set,

which are only a few, an *unknown item* vector was learned jointly. Therefore, 50 items were randomly sampled from the long-tail and masked with an *UNK* identifier when training the model. In particular, the sample set was $[\frac{id_{max}}{2}, id_{max}]$, as the items are sorted by frequency. Finally, this unique vector was assigned to all items which are not in the train set.

5.2 Online Test Simulator

In this section, the simulation environment is described that is used for online evaluation. At the time of writing, RecoGym [100] was the only publicly available simulation environment. However, it can not be initialized with given datasets but only be used with its own artificial data. Ie et al. [101] on the other hand tackled the challenge of a missing common evaluation simulator by proposing a recommendation framework. Unfortunately, at the time of writing, this is more of a software architecture framework, where one still needs to create every part for himself, instead of a ready-to-go simulation environment for comparable research on public datasets.

In this work, the user-behavior is modeled following the approach of Zou et al. [95]. In particular, the user-item-reward (or -interaction) matrix is completed by BPR-MF [102]. This is a widely used and established CF approach to estimate a feedback for all users and items. Hence, it seemed to be an appropriate approach to model the simulator this way. In practice, one computes the rating of item i for user u by $r_{u,i} = e_u^T h_i + b_u + b_i$. Hereby, e_u, h_i are learned representation vectors of dimension $\mathbb{R}^d$ for the user and item respectively. The bias vectors b_j are crucial because they capture user/item-specific rating abnormalities [103]. All vectors are learned jointly through gradient descent optimization.

The authors of [95] did not explicitly describe the chosen simulation algorithm. They converted the final matrix to probabilities through a logistic sigmoid-function, resulting in a lookup table of $P(click \mid u, i)$. However, there is no additional information about which particular item in the action list is chosen and, if no item from the list was selected, how the next state would look like.

Therefore, in this work, the matrix is standardized per row instead and the weights are directly used as rewards, just as they were intended by BPR-MF. The standardization centers the feedback around zero which will be beneficial when deciding if the user accepts an item or not.

Next to the reward matrix, it is also reasonable to model the user's patience into the simulator. This takes into account, that a user might get

Algorithm 3 User simulator

function STEP(Action a)
Require: User-index u, current state s_t, reward matrix R, original trajectory τ_u, current timestep t, maximal timestep t_{max}, previous prediction a_{t-1}, break probabilities p_{click} and p_{diff}, repetition factor ζ, action reject reward r_{reject}
  Get reward vector $r \in \mathbb{R}^{|a|}$ where $r_i = R[u, a^i]$
  `list-reward` $\leftarrow \frac{1}{|r|} \sum r_i$
  Filter a to only contain items with $r_i > 0$, same for rewards r
  $r \leftarrow r \cdot |$`list-reward`$|$
  Create $r_{none} \leftarrow max(r_{reject},$ `list-reward`$)$ and dummy item i_{none}
  Sample `next-item` from softmax distribution with weights CONCAT(r_{none}, r)
  Compute probability p_{break} for leaving according to Eq. (5.1)
  `done` $\leftarrow t = t_{max}$ **or** UNIFORM(0,1) $\leq p_{break}$
  if `next-item` $\neq i_{none}$ **then** ▷ User clicked on a recommendation
    `reward` $\leftarrow 1$
  else
    `next-item` $\leftarrow$ Next item in τ_u ▷ User follows his intrinsic goal path
    `reward` $\leftarrow 0$
  end if
  $s_{t+1} \leftarrow$ CONCAT(s_t, `next-item`)
  if `done` **then** ▷ Terminal states are not rewarded
    `reward` $\leftarrow 0$
  end if
  return s_{t+1}, `reward`, `done`
end function

annoyed and stops browsing the website due to bad or repetitive recommendations. In this work, a mixture of the simple approach of [75] and the item diversity-based model of [95] is applied.

Concretely, the probability that a user leaves the current session is modeled as

$$p_{break} = p(\text{break} \mid \cdot) + \zeta \cdot \textsc{Repetition}(a_{t-1}, a_t) \ , \tag{5.1}$$

where $p(\text{break} \mid \cdot)$ is determined by the user's reaction. If he interacts with a recommended item, the break probability is a predefined p_{click}, else p_{diff} for clicking on a different item. Further, repetition is defined as

$$\textsc{Repetition}(a_{t-1}, a_t) = \frac{|\{a_t\} - \{a_{t-1}\}|}{|a_t|} \ ,$$

which is the percentage of items that have been in the previous recommendation already. The hyperparameter $\zeta \in [0, 1 - max(p_{click}, p_{diff})]$ controls the impact of diversity punishment.

The simulator follows the interface convention of OpenAI Gym [104] and its STEP method is illustrated in Algorithm 3, where the recommendation is shown to the user as input and user returns a response. The created reward

matrix, which decides if an item from the recommendation is accepted, together with the probability function for leaving represent the environment's dynamics function as defined in Section 2.1.1.

The initial state of each episode is the item at a randomly selected time step in the user's trajectory. UNIFORM(0,1) is an auxiliary function that samples a scalar from the uniform distribution in the given interval and CONCAT($\vec{x}, y$) concatenates the scalar y to the end of vector $\vec{x}$.

At first, the rewards for each item in a are obtained from the created matrix for current user u . Next, the average reward over all items in the given action is computed. Additionally, all items that have a negative reward are filtered out. Then, the given rewards are scaled by the absolute value of the list-reward in order to penalize actions where many items are unsuitable. For example, it was found that an agent with a pure random policy has a list-reward around zero but also contains few items with higher reward. While the reward is given based on a specific item, the recommendation list should be pleasing as a whole as well. Without the list-average penalty, the simulator mostly accepted the one good item, leading to more return that it should have been given.

Afterwards, in Line 6, a dummy item for not choosing any of the recommended ones is created. Therefore, an artificial reward for this item is set to the maximum of a predefined constant r_{reject} and the list-reward. This is motivated by the following typical behavior: If the recommended items all have low rewards, the user is most likely to choose something else, unimpressed from the presented action. On the other hand, if all items are very popular, and hence have high reward values, the user might know about these already and might select another item. A stochastic component is introduced to these decisions by sampling from a categorical distribution initialized from the rewards.

If the user chooses not to interact with a recommended item, he continues his browsing journey with another item from his original recorded trajectory. The next state is created by concatenating the next click item to the previous state.

This procedure is repeated until the trajectory finishes, indicated by the `done` signal. Then, the next recorded user is selected, and the interaction with the RS begins from scratch.

Hyperparameter Settings. The probability of leaving even though the user clicked on a recommended item was determined to be $p(\text{break} \mid \textit{click}) = 0.05$ to somehow restrict the session length but also allow long-term strate-

gies. The diversity factor and probability for interacting with a different item were fixed at $\zeta = 0.2$ and $p(\text{break} \mid \text{no click}) = 0.3$ respectively. The first parameter was set to penalize repetitions not too much, because a user might simply not have acknowledged them, and the latter was determined through a parameter search to yield a low return for a random and a popularity based agent on ML-1M. These simple methods represent the extremes of possible RS policies. The random baseline produces diverse recommendations but they are completely uncorrelated to the user. As a result, the user's behavior should react to that and leave the current session due to not finding relevant products. On the other hand, the popularity agent will get a higher return as users interacted with popular items frequently. Due to the lack of individual and new recommendations, the user will also stop his session after a few clicks. However, the popular strategy should still receive a better return than random. The action reject reward is set to $r_{reject} = 1.5$ as it corresponded well with the previously defined parameters that keep the return of the two policies low.

This results in a calibrated simulator that responds to the recommendations of any RS in a natural way, i.e. it gives the highest cumulative reward for consecutive recommendations that have a good trade-off between diversity and personalization.

BPR-MF Implementation Settings. The Spotlight [105] framework was used to create the simulated ratings with implicit feedback BPR-MF. The dimension of the embeddings was 64 for all datasets with batch-size 256. The model has been trained for maximal 150 epochs and the model to create the final matrix was chosen manually by picking the first checkpoint of visible convergence. Furthermore, the learning rate was set to 0.001 and a weight-decay factor of 10^{-5} was applied. All other parameters are unchanged from Spotlight version 0.1.6.

5.3 Metrics

Following [106, 107], two kinds of metrics are employed to evaluate different aspects that a recommendation system should fulfill. Accuracy metrics measure how good a method can target the taste of a user as found in the recorded user trajectories. On the other hand, diversity metrics allow an insight to the predictions themselves. Does a policy actually select items from the whole item space or did it collapse to simply predicting the same popular items over and over again?

During offline evaluation, classification accuracy is captured by the *Hitrate@k* as it is a commonly seen metric in this area

$$Hitrate@k = \frac{1}{|D|} \sum_{\tau \in D} \frac{1}{|\tau|} \sum_{t=1}^{|\tau|} r_t \quad .$$

It measures the average number of correct predictions per user trajectory as binary rewards are used. In this setting, it is identical to the also popular Click Through Rate (CTR) [108]. In this notation, D is the set of users in the test dataset and τ is a user trajectory in it.

In some cases, the recommendation algorithm is prone to converge to a solution which recommends the same item list in many states. In order to capture this unwanted behavior, two diversity metrics [109] are collected as well. They express the general diversity found over the whole test-set. At first, *List-Diversity* describes the fraction of how many recommended lists were unique:

$$\text{List-Diversity} = \frac{|\text{unique actions}|}{|\text{all actions}|}$$

Two actions are seen as identical, if they contain the exact same items, regardless of their order. For example, $[1, 2, 3]$ and $[2, 3, 1]$ are subject to one unique list.
Additionally, the percentage of items of the full item set is reported that has been recommended at least once:

$$\text{Coverage} = \frac{|\text{unique items}|}{|\mathcal{I}|}$$

This is an important measurement in e-commerce, as the RS engine should recommend as much as possible from the available item catalogue to also sell niche products.

Other recommendation methods define diversity also w.r.t to items' content similarities [110] but in the given setting it is mainly of interest if the agents follow an adaptive and dynamic policy or if they have merely learned to propose a popular set of items, independently of the state.

As common in Reinforcement Learning, the mean *Return* is reported in the simulated online evaluation:

$$G = \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=1}^{|\tau|} r_t \ .$$

Due to the definition of the user simulator, this implicitly illustrates the trade-off between personalized recommendation and diversity.

5.4 Baselines

The outlined RL-methods are additionally compared to a set of commonly used baselines. The selection reaches from traditional methods, which are heavily used in Collaborative Filtering settings to another Reinforcement Learning approach:

- **POP**: A simple algorithm which predicts the most popular k items in the training set. Its predictions are independent of the current state.
- **ItemKNN**: One of the most popular CF-methods based on kNN and item-item similarities. Given a matrix that describes the interaction of users with items, one defines each item $i_j \in \mathbb{R}^{|\mathcal{U}|}$ as the interaction vector of all users $\mathcal{U}$ with item i_j. Each item representation is then weighted via TF-IDF and item similarity is computed according to the cosine-similarity. At inference time, the k nearest neighbors of the single most recently consumed item are recommended to the user. As the distance matrix can be computed in advance, inference is merely a lookup of the top-k elements and thus can be implemented in a highly efficient manner. Despite its simplicity, it was shown to be a very strong baseline [38], even outperforming complex neural algorithms.
- **GRU**: A simple GRU network that is trained with cross-entropy loss. It is basically the state-module of the RL-approaches followed by another hidden-layer and a Softmax output layer. The top-k list is created by choosing the most probable items of the output.This model represents the cross-entropy version of GRU4Rec [31], one of the most famous neural baselines, without the session-parallel batches. The GRU4Rec implementation was not used directly due to an interface mismatch for using a custom simulator. For better comparability with the RL approaches, GRU also exclusively uses Word2Vec item representations.
- **DUELING**: Another Q-learning RL-algorithm with a Dueling architecture [85] that predicts a value for each available action.

5.5 Experimental Setup

The majority of hyperparameters is fixed over all evaluation settings. The last $m = 10$ items that a user has interacted with form the state. For offline evaluation, the initial state s_0 consists of the first item of the trajectory and increases with every interaction up to the last m items . One action

recommended by the policy consists of $k = 10$ items. During testing, all trajectories are limited to 100 interactions. One episode in this scenario is defined as interacting with 32 users simultaneously over their whole trajectory. Every 500 episodes, or after one dataset iteration in case of ML-1M, a method is applied on the validation set and a model checkpoint is saved to disk. The agent receives a reward of 1 when the environment accepts one of the items from the recommended list, else 0.

The dimension of embedding vectors is set to a fixed size of 16. This was selected through a small experiment on ML-1M with NEWS where no performance gain was achieved by using 32 or 64 dimensions, while it did lengthen training time significantly. Actor-Critic methods use normalized embeddings in the interval $[-1, 1]$, corresponding to a *tanh* output layer ot the actor. Without these bounds, the policies performed poorly and suffered from exploding weights almost instantly. Network hyperparameters were chosen according to the ones provided in the respective paper. All values that were not described, which are the majority, were set to values which were found to be of reasonable value manually. Due to the unfeasible number of theoretical runs for an exhaustive hyperparameter search, there was no other choice than selecting those from the respective work or following intuition. All neural network layer sizes are approximately in the same range of total number of parameters.
If appropriate, target networks are updated after every transition in a moving average manner through polyak averaging $\theta' \leftarrow (1-\tau)\theta' + \tau\theta$, as it is common in the field. Here, θ' are the parameters of the target network and θ are the current parameters of the most recent policy. The impact factor was set to $\tau = 0.005$ which follows most work in literature.
The discount factor of the MDP was primarily set to $\gamma = 0.99$, i.e. future rewards are weighted heavily. Gradient clipping at 5 was applied for a more stable training procedure. All methods are optimized by Adam [111] with default values for the momentum variables and a learning rate of $\eta = 0.001$. Policy-Gradient methods improved with a L2 regularization of $\lambda = 1e - 4$. This was also used for off-policy approaches in the first evaluation but was then found to hinder performance in the second. Therefore, it is set to 0 in the offline evaluation from batch.

The GRU-cell of the state-module has a dimension of 128 units and the last hidden state is used as the state representation. ReLU [25] was selected as activation function for all hidden layers. Policy-Gradient methods use a softmax function for their final probability distribution. Neural network hyperparameters are summarized in Table 5.2. REM and QR-DQN use $K = 25$ heads on the output. This means, REM leverages the mean over 25 Q-values and QR-DQN separates the estimated value distribution into 25

Method	Actor	Critic	L2-factor	Output function
DUELING	-	[200, 200]	10^{-4} (0)	-
GRU	[128]	-	10^{-4}	Softmax
CORRECTION	[256]	-	10^{-4}	Softmax
TPGR	[32, 25]	-	10^{-4}	Softmax
WOLPERTINGER	[50, 50]	[100, 100]	10^{-4} (0)	TanH / -
LIRD	[50, 50]	[100, 100]	10^{-4} (0)	TanH / -
NEWS	-	[200, 200]	10^{-4} (0)	-
REM	-	[200, 200]	0	-
QR-DQN	-	[200, 200]	0	-

Table 5.2: Neural network hyperparameters for all methods and baselines. The numbers in square brackets refer to the number of units per hidden layer. The output functions for Wolpertinger and LIRD refer to actor and critic respectively. The weight decay factor was set to 10^4 for off-policy approaches at first, but has shown to restrict training in the second evaluation setup. Therefore, L2 regularization is only used in the first evaluation setting.

fractions.

Off-policy algorithms, i.e. every method that involves a Q-network, uses a prioritized experience replay [67] with a buffer size of 100k. The priority constant α and the importance factor β are set to a fixed value of $\alpha = \beta = 0.5$. Those choices were inherently motivated by the original publication. When sampling from the replay buffer, a batch size of 64 was used. There has been no information found in any publication about which items from an action are stored in the replay buffer. Therefore, with exception of LIRD, all off-policy methods *only* store the item for which it received a reward or the first item of the list, in case the user did not click on any recommendation. Furthermore, the transition tuple (s, a, r, s') stores the *item ids* in the states and action respectively. This means, when drawing a tuple for updates, the *latest* state module (and embedding) is leveraged, as it is seen as part of the actor. This information was not found in any other publication but it seemed more reasonable than to store the state *representation* that will have changed drastically at the point when a tuple is drawn from the buffer.
In the first evaluation setting, the methods follow an ϵ-greedy exploration policy, set to a fixed $\epsilon = 0.25$. Additionally, supervised pretraining was performed for 3000 batches of size 64 in this setting. Actor Critic approaches update the state module exclusively together with the actor. During critic optimization, the state is treated as a constant, following [95].

For Policy-Gradient methods, a Q-value normalization per user trajectory is applied when calculating the explicit discounted return as a baseline in REINFORCE. This is common practice in order to keep the values at scale, which is a desirable property when working with neural networks. The importance weights used for off-policy compensation are clipped at 10.

All methods were implemented in Python using PyTorch[3]. Each experiment is executed on a Kubernetes cluster with limitations to 4 CPUs, 8 GB memory and one GPU. Due to the variety of different GPUs in the local cluster, the job is computed either on a NVIDIA GeForce GTX 1080Ti, GeForce RTX 2080Ti or Titan X. Main experiments are run for three random seeds and the mean and standard deviation for each algorithm and metric will be reported. All error bars or bands in figures represent a 95% confidence interval.

5.6 Results

This section outlines the results for each evaluation setting respectively. The first setup applies RL directly on the log data, where agents need to explore heavily to get a reward for a prediction. Then, the agents are trained in an offline RL setting, where a model has direct access to the collected state transitions of the dataset. As the correct next items are given for the test dataset, classification metrics can be used to measure performance. Lastly, in order to consider recommendations of unseen transitions, each agent is evaluated on the online simulator w.r.t its average cumulative reward.

5.6.1 Pure RL on Log-Data

As a first evaluation, all methods are trained by following the user trajectories which are given by the dataset. For this experiment, only ML-1M is used. If the methods performed poorly on this small dataset, they most likely perform even worse on a more sparse dataset such as Taobao. CORRECTION is not included as its contribution lies in the guided offline setting from batch. In this evaluation, TPGR has no off-policy compensation through IS because it would allow an information advantage to the other methods. REM and QR-DQN are also excluded as the performance of established approaches should be evaluated first to test this setting. The results of all methods and baselines are shown in Table 5.3.

[3] `https://pytorch.org/`

Experiment	Method	Hitrate	List-Div.	Coverage
Baselines	POP	0.051	0.000	0.003
	ItemKNN	0.127	0.040	0.956
	GRU	**0.159 ± 0.002**	0.990 ± 0.000	0.854 ± 0.032
Scratch	LIRD	0.006 ± 0.002	0.000 ± 0.000	0.002 ± 0.000
	Wolpertinger	0.007 ± 0.003	0.000 ± 0.000	0.002 ± 0.000
	DUELING	0.008 ± 0.002	0.001 ± 0.000	0.010 ± 0.001
	NEWS	0.047 ± 0.001	0.012 ± 0.019	0.016 ± 0.013
	TPGR	*0.073 ± 0.006*	0.431 ± 0.058	0.081 ± 0.065
Word2Vec	LIRD	0.009 ± 0.002	0.000 ± 0.000	0.004 ± 0.002
	Wolpertinger	0.042 ± 0.001	0.000 ± 0.000	0.003 ± 0.000
	DUELING	0.011 ± 0.004	0.001 ± 0.001	0.008 ± 0.001
	NEWS	0.044 ± 0.006	0.004 ± 0.006	0.012 ± 0.005
	TPGR	*0.094 ± 0.002*	0.576 ± 0.004	0.180 ± 0.232
Full	LIRD	0.007 ± 0.001	0.000 ± 0.000	0.006 ± 0.004
	Wolpertinger	0.043 ± 0.001	0.000 ± 0.000	0.003 ± 0.000
	DUELING	0.005 ± 0.001	0.053 ± 0.040	0.041 ± 0.018
	NEWS	0.059 ± 0.003	0.216 ± 0.219	0.011 ± 0.091
	TPGR	*0.112 ± 0.001*	0.663 ± 0.054	0.036 ± 0.019

Table 5.3: Experimental results of the first evaluation setup on ML-1M with different levels of pretraining. Only on-policy TPGR was able to learn good policies, while still performing worse than the ItemKNN baseline. Experiment *Word2Vec* added pretrained item representations and *Full* refers to supervised pretraining of all neural networks.

Baselines. The supervised GRU model is obviously the best, while having low-variance over three independent seeds. Comparing ItemKNN's and GRU's diversity metrics, the difference between the two metrics can be illustrated. In particular, ItemKNN recommended around 96% of available items at least once over the whole test set, indicated by the coverage. Still, most of the time, the method repeats its recommendation list multiple times, as illustrated by the List-Diversity. GRU on the other hand manages to receive the highest return of all baselines while having much more diverse recommendations in its composition.

RL Agents. Next, the performance of RL-agents is analyzed. At first the results of the experiment without pretraining is assessed before evaluating the effect of different levels of pretraining.

Already on the first glance, it is observable that none of the approaches are competitive to the baselines. The on-policy approach TPGR is the best

of the reinforcement learning approaches but its diversity metrics are significantly worse than those from the static baselines. All other off-policy methods fail completely to learn a policy. This is not only indicated by the Hitrate but also by the barely existing List-Diversity. All of these policies collapse to a deterministic prediction of the same few items which have lead to a reward through exploration during training. As Wolpertinger, LIRD and NEWS directly work in the feature space, maybe the randomly initialized item representations are the cause for this performance. Since the models barely receive a positive reward, the main value estimate is around zero, just as the TD-Error at some point. This small error is then the only signal to optimize those representations through backpropagation, which decreases its magnitude even further through all layers and can not enhance the embeddings significantly.

Therefore, pretrained Word2Vec vectors are used in a follow-up experiment to investigate this assumption. This improves the performance of Wolpertinger by more than 3% but its diversity remains poor. The other off-policy methods do not show any significant change to the previous experiment. TPGR, on the other hand, can boost its performance by 2 percent and also its diversity. However, it still can not achieve a competitive Hitrate to the static ItemKNN.

A final experiment is conducted to see the impact of full supervised pretraining. This was assumed to drastically boost the performance of each approach as they are optimized directly towards a correct target. Contrary to these expectations, TPGR is the only approach, again, that increases the quality of its recommendations. However, it still can not reach the performance of the baselines. These, however, have had access to way more correct state transitions during training while TPGR had to explore with a stochastic policy to find new promising items.
Thorough investigation of the off-policy algorithms' training procedure revealed that the validation performance did increase in the course of pretraining. However, after only a few episodes of RL, the policies completely diminish. LIRD and DUELING additionally suffer from value overestimation almost immediately. All methods take around 8 to 12 hours for 10,000 episodes and evaluation on the valid dataset every 500 episodes.

In conclusion, it becomes apparent that this experimental setup is unsuitable to effectively train a competitive recommender system with any of the RL methods. Leveraging Word2Vec embeddings enhances some algorithms while full supervised pretraining only increases the performance of TPGR. Additionally, LIRD could not be pretrained efficiently with the given data due to its explicit list creation. All RL agents have demonstrated poor item coverage, i.e. they exclusively repeat the popular items that have been en-

countered several times.

5.6.2 Offline Evaluation

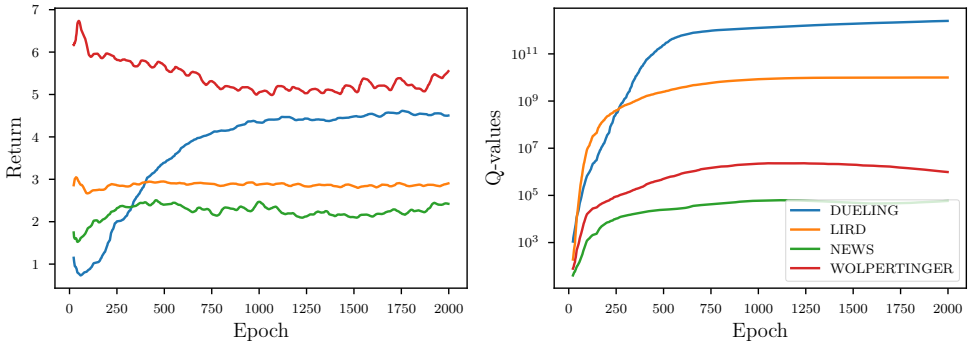
Next, the idea of offline RL is followed where all methods are trained with their RL procedure. This means the approaches are leveraging the offline logs (train fraction) directly to develop a policy. Although, before full evaluation of all methods, the batch learning procedure had to be adapted to battle value overestimation of value-based algorithms, i.e. all approaches but TPGR and CORRECTION.

The Recurrent Problem of Overestimation

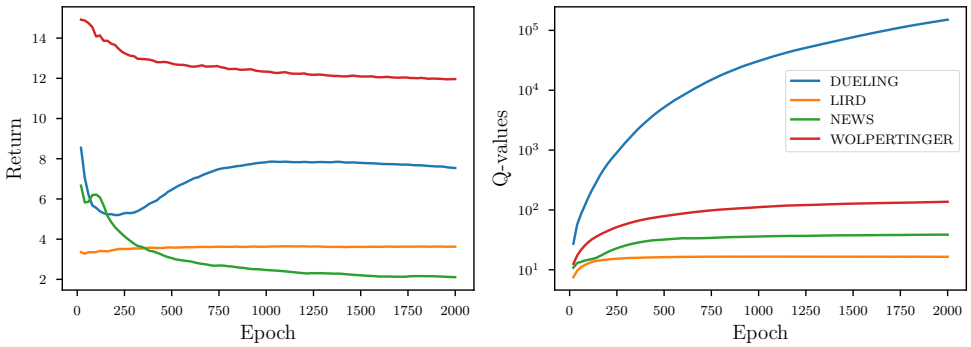
As introduced in Section 2.1.2 of this work, the Q-value is supposed to estimate the expected return in a particular state. This means, possible value estimates are in the interval $[0, t_{max}]$, as here the maximal immediate reward is 1. Further, the most realistic values are assumed to be in the lower part of this interval, e.g. $[0, 10]$, as the behavior policy that lead to the batch of trajectories most likely followed a myopic strategy.

Figure 5.3a shows the performance during training for all off-policy methods in the first 2,000 episodes. On the left, the moving-average of the return is illustrated while the respective estimated Q-values are shown on a logarithmic scale on the right. Contrary to theory, all methods immediately start to overestimate the true action-value by far. Recall, that the loss-objective for these Q-networks is the mean-square TD-Error. This means, the error is the squared value of these action-values, as the target values are initialized around zero at the beginning. As a consequence, the gradients of the learned parameters are exploding as well, which is definitely undesirable in deep learning. One of the reasons why the models do not crash instantly due to numerical issues is the application of gradient clipping.

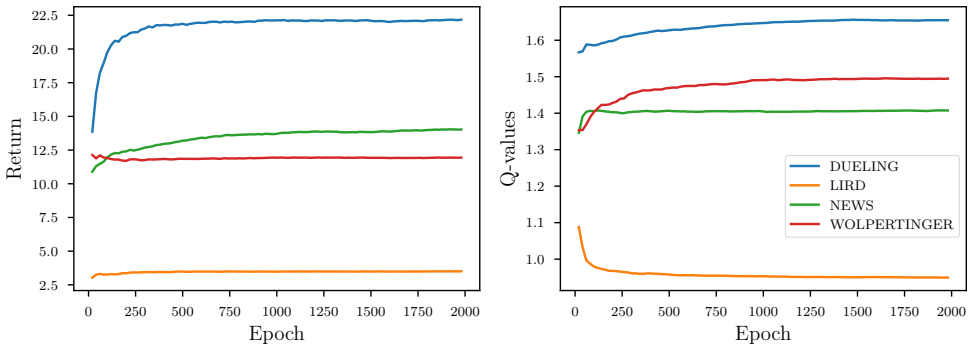
How can DUELING start to increase its return after around 100 episodes, even though its value estimation is totally off? One explanation might be the following. It is the only off-policy method that has got one action value output for each available item. This means, the more frequently the item occurs in the trajectory, the more weight it gets to raise this item's value. This quickly leads to a deterministic policy of recommending some of the most popular items. This behavior is also independent of the state as these frequent items get the highest value in the output layer by far, fully neglecting smaller differences in preceding layers. This hypothesis is supported by closer examination of the recommendations: Most lists are completely identical and include only small item ids, e.g. items 2, 3 and 4.



(a) Drastic overestimation in first Batch-RL experiment.



(b) The result of adding negative examples for overestimation reduction.



(c) Reducing the discount factor to stronger neglect future rewards.

Figure 5.3: Training process of off-policy methods for the Batch-RL setting. On the left, the average return per episode and user batch is shown. On the right, the corresponding mean Q-value expectations of the first step of an episode is illustrated. The value estimates are continuously reduced by first (b) adding negative examples and then (c) reducing the discount-factor. Notice the change of the y-axis scale to linear in (c).

Before evaluating these algorithms on all datasets, this drastic overestimation needed to be contained first. Until now, the replay buffer, from which the Q-network samples from in order to improve the policy, was successively filled with true transition tuples $(s, i_t, r(i_t, s) = 1, s')$ from the dataset. This means that it is good $(r = 1)$ to predict item i_t in state s and this will lead to s'. For example, given state $[1, 2]$, the next target item 3 and the resulting next state $[1, 2, 3]$ is added as correct transaction to the buffer. As each observed transition in the buffer leads to a reward, the network becomes more and more optimistic in all its value projections.

As a countermeasure, the policy's prediction for a state $(s, i_\pi, r(i_\pi, s), s')$ is added to the buffer as well, additionally to the correct transactions from batch. As always, i_π is the item that was clicked by the user or the item on top of the recommendation list. Continuing the example from above, the agent recommends $[4, 5, 6]$ in state $[1, 2]$ to the user and receives a zero reward, together with the next state $[1, 2, 3]$. Then the tuple $([1, 2], i_\pi = 4, r = 0, [1, 2, 3])$ is also added to the buffer.

This introduction finally reduces the overestimation by adding negative examples because not every prediction is correct, as it was indicated before. At some point, the policy will then get no reward in a state for an action that always lead to a reward in other states. Through this, the popular actions' values are penalized which leads to a higher state affinity and finally to more achievable return, as shown in Fig. 5.3b. The only method remaining with a drastic overestimation is DUELING, even though the impact was already decreased by multiple orders of magnitude. The other three methods are now estimating values in a realistic magnitude with values smaller than t_{max}, which was limited by preprocessing to $t_{max} = 300$ for this dataset. The performance of Wolpertinger and even DUELING clearly benefited from this measure. NEWS started much better than before but than completely collapsed. Note, that the illustrated performance in Fig. 5.3 is measured during training, hence these metrics are not comparable to any value before.

As those values still appeared way to high for this recommendation setting, the investigation for an even better solution continued. Therefore, a small hyperparameter study was executed to research the impact of the one factor that is directly involved in the value estimation - the discount factor γ. Until this point, it was set to $\gamma = 0.99$, i.e. all future value estimates are contributing almost completely to the current estimate. By running each method once for all values between $[0.1, 0.9]$ in steps of 0.1, $\gamma = 0.6$ was found to work much better on ML-1M. The enhancement is clearly visible in Fig. 5.3c. The value estimates are realistically small, which enhances stability of the models' training procedure. Additionally, an underestimating value function, i.e. a pessimistic one, is not as much of a problem as one that

overestimates each value [18]. This effect directly translates to much better return values. As this new discount factor is obviously more appropriate for this dataset, it is also adapted by the on-policy methods for the following evaluation.

Despite these corrections, the performance of Actor-Critic approaches Wolpertinger and LIRD could not be enhanced. Especially LIRD is failing completely to learn anything in this batch setup. The main reason is because it tries to recommend a whole list and then gets a reward for that. However, the reward signal is averaged by the whole output during gradient computation, which makes learning through the policy gradient even more difficult, if not impossible.

On a final note, it is also worth highlighting how quick the off-policy methods actually converge. After 500 episodes, which are around 5 dataset iterations, the Q-value approaches have reached convergence. However, this fast stop of improvement seemed suspicious, as Wolpertinger could not further enhance its policy after 20 episodes and DUELING also fully converged after only 3 iterations. An investigation of shared hyperparameters then revealed that the problem is the L2 weight decay factor. Even though the value was chosen relatively small already with $\lambda = 10^{-4}$, it appeared to restrict the value-based agents in their optimization. For this reason, all off-policy approaches do not leverage L2 regularization in the following evaluation on all datasets. Overfitting is not a problem after all, as the final model is selected from regular checkpoints based on the performance on the validation dataset.

Offline Evaluation Results

Finally, all methods could be trained from batch and the results are summarized in Table 5.4. For this setting, the larger datasets are included as well for a more general overview. As the target items are given by the test-set, Hitrate@k can be computed together with diversity measures. Each non-static method was run for three different random seeds. The difficulty increases for each dataset as the action spaces become larger and sparser. At first, the results are analyzed in general before each model is examined individually.

The overall observation is that no single RL approach stands out from the rest. Depending on the dataset, the ranking of the methods changes completely. This highlights that the choice of datasets is always critical for comparison. Furthermore, no clear difference between on- and off-policy algorithms with respect to accuracy is visible in general. The coverage of on-policy methods, however, is consistently smaller than the one from off-policy approaches which is undesirable in practice. The list-wise approach

(LIRD) was unable to optimize its policy at all. The two promising methods, QR-DQN and REM, fall short of their expectations and do not outperform the other classic approaches.

Baselines. The myopic GRU baseline performs very good on all three datasets, outperforming all RL approaches. ItemKNN again proves itself to be a strong but simple baseline, especially on Taobao. In addition, it consistently shows a coverage of over 94%. This is a desirable value, as companies also want to recommend, and finally sell, niche products instead of merely popular items. The discrepancy between list diversity and coverage of ItemKNN can be understood as follows: Each item has its k nearest neighbors and nearly each item is obviously involved in another item's list. This explains the high coverage of the approach. As items appear more frequently, the identical list is recommended repeatedly. This drastically decreases the diversity of lists over the whole test set.

LIRD. The obviously worst agent is LIRD [7], that does not even get close to the performance of the popular baseline. The competitive accuracy to NEWS, as outlined in the original publication, was not achieved at all. In contrast, the agent was unable to learn any kind of policy. One of the reasons for this is that the list size of $k = 10$ might simply be too big. In the publication, an evaluation of the list size determined to work best with a maximum of $k = 4$. However, recommending only four items at a time seems very restricting in its use case, which is why $k = 10$ was chosen for this evaluation after all.
Furthermore, in the offline setting, the actor is updated with the MSE between the next click and the proto-item at the first list-position in addition to the policy gradient. As the policy gradient estimate is computed through the critic, the critic itself needs to be well-trained. However, it clearly can not compute good enough value estimates for a whole item list, which is the input of the critic. Consequently, as neither a good value approximation nor a good actor could be learned, the policy learning can be considered as failed.

Wolpertinger. Wolpertinger [75], which was not compared to any baseline originally, can establish a working policy in this setup as well. On the MovieLens datasets, however, its performance is on the lower half of the ranking of all approaches. On the other hand, it works surprisingly well on Taobao. In addition, it can be observed that this approach is the most diverse with a consistent list diversity of over 91%. This is surprising as it was rather expected that the proto-item would be computed simply in the center of a

Data	Method	Hitrate	List-Div.	Coverage
ML-1M	POP	0.051	0.000	0.003
	ItemKNN	0.127	0.040	0.956
	GRU	**0.192 ± 0.001**	0.785 ± 0.028	0.556 ± 0.050
	LIRD	0.018 ± 0.002	0.617 ± 0.040	0.333 ± 0.018
	Wolpertinger	0.130 ± 0.002	0.982 ± 0.007	0.844 ± 0.002
	DUELING	0.142 ± 0.002	0.786 ± 0.020	0.610 ± 0.015
	NEWS	*0.159 ± 0.006*	0.899 ± 0.060	0.431 ± 0.015
	TPGR	0.123 ± 0.001	0.274 ± 0.014	0.030 ± 0.035
	CORRECTION	0.151 ± 0.003	0.467 ± 0.028	0.090 ± 0.010
	QR-DQN	0.130 ± 0.002	0.740 ± 0.003	0.354 ± 0.012
	REM	0.147 ± 0.005	0.952 ± 0.022	0.517 ± 0.048
ML-25M	POP	0.060	0.000	0.000
	ItemKNN	0.151	0.007	0.946
	GRU	**0.247 ± 0.002**	0.640 ± 0.031	0.168 ± 0.003
	LIRD	0.013 ± 0.001	0.208 ± 0.054	0.157 ± 0.025
	Wolpertinger	0.175 ± 0.001	0.911 ± 0.018	0.587 ± 0.057
	DUELING	0.184 ± 0.009	0.517 ± 0.059	0.133 ± 0.022
	NEWS	0.193 ± 0.022	0.757 ± 0.120	0.343 ± 0.117
	TPGR	*0.220 ± 0.002*	0.336 ± 0.013	0.111 ± 0.008
	CORRECTION	0.202 ± 0.007	0.345 ± 0.014	0.018 ± 0.002
	QR-DQN	0.131 ± 0.010	0.172 ± 0.033	0.053 ± 0.005
	REM	0.177 ± 0.010	0.821 ± 0.066	0.443 ± 0.126
Taobao	POP	0.008	0.000	0.000
	ItemKNN	**0.098**	0.077	0.997
	GRU	0.077 ± 0.001	0.968 ± 0.002	0.511 ± 0.006
	LIRD	0.004 ± 0.000	0.834 ± 0.088	0.268 ± 0.014
	WOLPERTINGER	*0.029 ± 0.006*	0.986 ± 0.002	0.674 ± 0.095
	DUELING	0.020 ± 0.001	0.970 ± 0.002	0.191 ± 0.018
	NEWS	0.020 ± 0.001	0.948 ± 0.011	0.111 ± 0.014
	TPGR	0.022 ± 0.001	0.277 ± 0.108	0.053 ± 0.002
	CORRECTION	0.014 ± 0.011	0.189 ± 0.327	0.005 ± 0.008
	QR-DQN	0.004 ± 0.000	0.907 ± 0.052	0.087 ± 0.036
	REM	0.021 ± 0.001	0.945 ± 0.026	0.148 ± 0.023

Table 5.4: Experimental Results for all methods on different datasets. The ranking of RL-based approaches is nearly consistent over the datasets. GRU is a strong baseline, uniformly outperforming all RL-methods. The best method in total is highlighted in bold, while the best RL-approach is highlighted in italics. TPGR and CORRECTION are on-policy and the rest off-policy approaches.

popular cluster.
As this approach directly computes proto-items in the feature space, the quality of item representations is crucial. In this evaluation, only contextual features in form of Word2Vec embeddings were used, instead of content information. It is assumed that more advanced embeddings can enhance the overall performance of Wolpertinger.

NEWS. The approach with the dueling parametric DQN [84] is better on average over all three datasets than the non-parametric variant DUELING. This architecture obviously benefits from similarities between items in the feature space for a more robust value function. However, since each action representation must be evaluated together with a copy of the current state to compute a value for each action, its practical use case is limited. Because of this additional overhead, the method was much slower w.r.t to wall-clock time in comparison to other methods. Hence, this approach is only feasible if the item-space is of manageable size or another component subsamples a candidate set beforehand.

TPGR. The Tree-based policy gradient approach is not as much better than Wolpertinger or NEWS as it was shown in the paper [74]. Contrarily, the both other approaches are actually more accurate on ML-1M and Wolpertinger clearly outperforms TPGR on Taobao. Unfortunately, ML-10M, yet another MovieLens subset, was used in the original evaluation which does prohibit further comparison to this study. It is surprising anyway that this mid-size subset was selected, as the main contribution is handling large action spaces, which are available in ML-25M and Taobao. The introduced speedup with respect to wall-clock time is also necessary, as Fig. 5.4 shows that much more iterations are needed until convergence. The figure might suggest that more episodes would have further improved the performance. However, this was found not to be the case in practice. It is also worth to highlight that even though the Hitrate is comparative to the other RL approaches, especially the item coverage is poor. On ML-1M and Taobao, the approach recommends less than 6% of available products. This could probably be enhanced by taking advantage of a stochastic policy during testing and introducing entropy regularization in the loss function to prevent a collapse to a deterministic policy [112].

CORRECTION. The on-policy agent with an additional off-policy correction factor could not validate the impact of its contribution [6]. Especially on Taobao, the method performed worse than the previously analyzed RL

agents, with exception of LIRD of course. However, the standard deviation on this dataset is large, so one of the three random seeds was apparently suboptimal. In addition to that, its item coverage between 1 and 10 percent is very low, i.e. it also collapsed to a rather deterministic policy of popular items. However, it can not be deduced from this evaluation if the introduction of the correction factor λ enhanced the performance of the traditional REINFORCE agent. Therefore, a version without the top-k correction and with the same architecture would have been necessary.

QR-DQN and REM. For the first time, two more recently off-policy methods were evaluated on these recommendation datasets. Originally, they were shown to drastically improve the performance on Atari games in offline RL, i.e. when an agent is restricted to a batch of trajectories and no direct interaction with the environment is possible [71, 91]. Unfortunately, both algorithms did not surpass the performance of the classic Q-Learning procedures. QR-DQN [91] was even unable to learn a policy at all on Taobao, sharing last place with LIRD. In addition, it was not expected that REM [71] performed worse, on average, than NEWS because REM is practically merely an ensemble version of NEWS. The authors conducted a study of the needed number of interaction tuples and showed that both algorithms required at least 5M interactions to be comparable to an online agent, preferably more [71]. Maybe the sparsity of the datasets, where not many actions are covered for possible states, is the problem why they can not repeat their superiority in this domain.
Both approaches would have been worthy of consideration in the same application scenario as NEWS, i.e. if another model provides a candidate subset of the full item space. This can be found in multiple production setups, e.g. at YouTube [6]. They are less ideal to cover the *whole* item space because they compute not only one value per item but K. Nevertheless, it is interesting for the research domain that these algorithms do not add a benefit compared to the classic ones in usage.

Additional observations

Figure 5.4 shows a moving average of the Hitrate during training on the held-out validation trajectories of ML-1M. The first interesting observation is the very first high value of the off-policy algorithms. After only one iteration of the dataset, all approaches have already achieved a Hitrate of at least 8%. This kind of data-efficiency is typical for off-policy methods and highly desirable in practice. Recommender systems need to be trained continuously as new items are introduced or new user trajectories have been col-

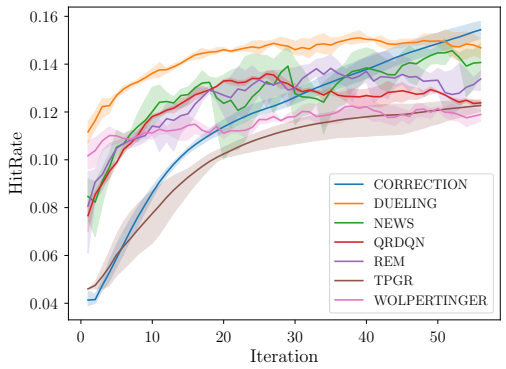


Figure 5.4: Validation accuracy during training on ML-1M.

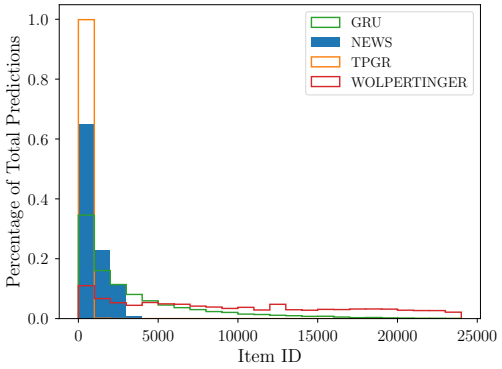


Figure 5.5: Distribution of predictions on Taobao.

lected to improve the model. Differently, the two on-policy methods TPGR and CORRECTION need much more data until convergence, even though the performance surpasses off-policy agents in the long run eventually. The trend of CORRECTION even indicates that more iterations could have further improved the model.
The heavy delay of convergence could also be subject to the behavior policy approximation. In the beginning, the behavior model β is completely random and has to improve first before the target model π_θ can leverage the probability estimation for its importance weights $\frac{\pi_\theta(a_t|s_t)}{\beta(a_t|s_t)}$. As the approximation gets better, the policy gradient estimates become more exact due to better off-policy compensation. This hypothesis could be validated by using a fully-trained model from the beginning, e.g. by leveraging the GRU baseline, but it is left for future work as it is not in center of this work's goal.

As an additional insight, the coverage metric for the best performing methods of each type on the Taobao dataset is illustrated in Fig. 5.5. The predictions of Wolpertinger are nearly uniformly distributed over the whole catalogue of 25,000 items, while also being the most accurate RL approach. The by far best method, GRU, has got a definite tendency towards the more popular items, however the prediction distribution also has got a long tail with niche products. NEWS on the contrary only predicted actions with the most popular 4000 items. This was only outdone by TPGR which became a pure mainstream recommender where around 99% of its predictions are within the top 1000. This illustration and the usage of diversity metrics emphasizes that a recommender system should not be evaluated and finally selected for productive application exclusively through accuracy metrics as it can be found frequently in literature [7, 78, 74].

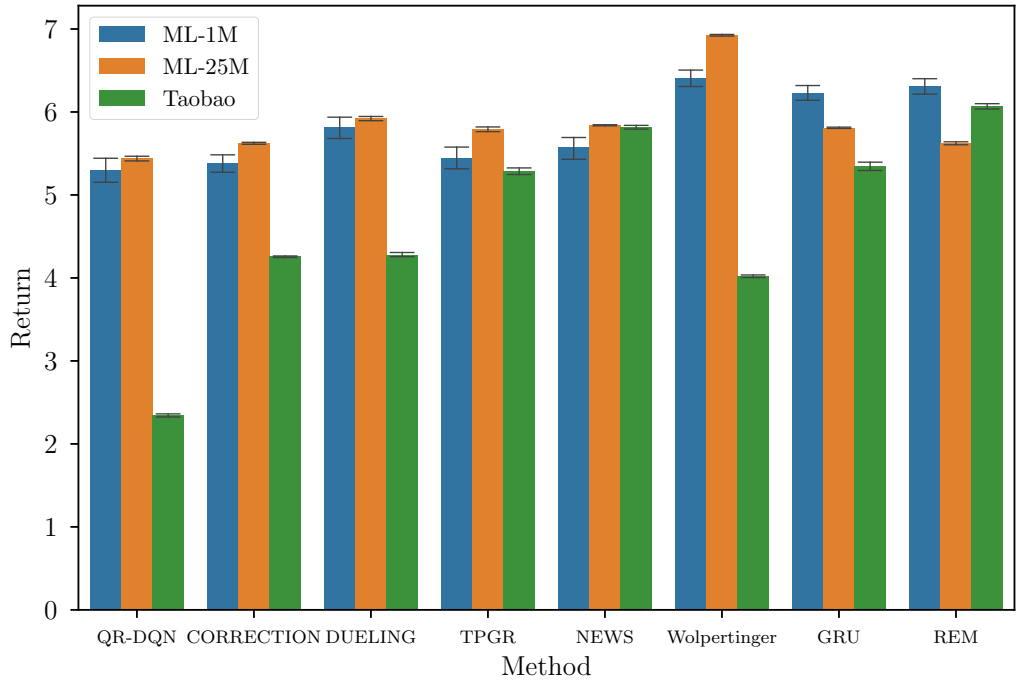


Figure 5.6: Average user return per method and dataset. Each agent was run five times on the simulator with different random seeds and the mean return is illustrated with error bars for the 95% confidence interval.

5.6.3 Online Evaluation

In order to close the gap between the selected action of an agent and the next state, the test data is used at this point to create a simulator. This allows an agent to receive rewards for items where no interaction was available in the logging data. Therefore, all pretrained models from the previous setting are reused and evaluated on the artificial online environment.

Results

The performance of all agents in the simulated online environment is illustrated for each dataset in Fig. 5.6. The approaches are sorted by their average return over all seeds and datasets, even though it is barely visible. At this point it is sufficient to only report the average undiscounted cumulative rewards per user as this directly reflects a model's diversity-accuracy trade-off through the simulator's defined behavior (see Section 5.2). The figure shows the mean performance of five simulation runs with error bars indicating a 95% confidence interval. LIRD is not part of this online evaluation because

it was shown to not learn a policy at all in the previous setting.

The ordering by the mean return obviously does not reflect the performance for each dataset respectively. Rather, it can be observed that most methods perform equally over certain datasets and there is no clear 'best' method, similar to the previous offline setup. As this figure illustrates a lot of information, it is important to rather focus on results that deviate from offline evaluation and are hence unexpected.

The main result in this evaluation is that introducing RL to optimize for a long-term reward did not show a clear advantage over a myopic strategy, followed by GRU. However, in the offline evaluation, GRU was always significantly better than RL-approaches, up to 250% of the best RL approach on Taobao. Here, on the other hand, it merely performs competitively to the RL methods and can not achieve the first rank on any dataset.

Another interesting fact is that the model which performed best on a dataset in the previous setting, can not repeat its superiority in the online application. For example, Wolpertinger was clearly the best RL agent on the Taobao dataset with respect to accuracy *and* diversity. In the simulation, it only ranks on the second last place, after the randomly behaving QR-DQN. This is probably due to its high coverage in comparison to other methods. Wolpertinger recommends more niche items than other agents and these items are barely included in the test set due to its sparsity. Hence, no good representations were built by the BPR-MF routine of the simulator which leads to less cumulative reward after all. The same observation, that the previously best method here performs worse than others, holds for NEWS on ML-1M and TPGR on ML-25M, too, even though the discrepancy is smaller.

REM has got the highest mean return over all datasets. This is mainly due to its highest performance on Taobao, the sparsest dataset with the largest item space. Even though the difference between the return of NEWS with 5.8% and REM with 6.1% sounds marginal, it can lead to a large increase of the profit of an e-commerce store. However, as already mentioned in the previous subsection, due to the computation of multiple Q-values per state, it took over 24 hours to train the method for 15,000 episodes on Taobao. For comparison, Wolpertinger only took around 10 hours on average.

Qualitative Insight

After observing the average return for each method in an online scenario, there was a curiosity about the actual recommendations behind the numbers. Therefore, one recommendation example is illustrated in Fig. 5.7. It shows recommendations created by GRU, TPGR and Wolpertinger on ML-25M

State	Titanic
GRU	Toy Story (11), **Star Wars: Episode VI** (15), **Star Wars: Episode IV** (6), **Star Wars: Episode I** (99), Independence Day (25), **Star Wars: Episode V** (14), Pulp Fiction (3), The Silence of the Lambs (4), Schindler's List (6), Indiana Jones and the Raiders of the Lost Ark (18)
TPGR	Toy Story (11), Back to the Future (31), Jurassic Park (7), **Star Wars: Episode IV** (6), The Matrix (5), **Star Wars: Episode I** (99), **Star Wars: Episode V** (14), The Princess Bride (44), Independence Day (25), Indiana Jones and the Raiders of the Lost Ark (18)
Wolpertinger	Independence Day (25), Twister (67), Sense and Sensibility (129), Broken Arrow (133), Executive Decision (291), Turbo: A Power Rangers Movie (3684), There's Something About Mary (155), 'Bell, Book and Candle' (3230), Air Force One (419), Dolores Claiborne (604)

Figure 5.7: Exemplary recommendation from GRU, TPGR and Wolpertinger when predicting suitable next movies after the user consumed *Titanic*. The predictions where obtained on the ML-25M simulator and numbers in parentheses represent the respective movie ids in the dataset.

respectively. Each agent's action is based on the implicit feedback that the user has just interacted with the famous movie 'Titanic'. The movie ids are included to see if a method only chooses popular movies or ones that actually correspond well to the current state, even if that means a riskier prediction has to be made.

The only recommendation that these 3 models have in common is 'Independence Day', a disaster/science fiction movie. GRU and TPGR share 6 out 10 predictions and all of them are not connected in any sense to Titanic. On the contrary, both methods even include three to four Star Wars movies in their recommendation list. This was not a random incident, in fact, an analysis of the TPGR evaluation predictions revealed that nearly 60% of all actions involved at least one Star Wars movie. One could say that the force towards these movies is strong in TPGR.

A different behavior is found in the recommendations of the Wolpertinger agent. As already indicated by the movie ids, the method does not exclusively rely on popular choices but also adds rather niche items. In contrast

to the other two models, it recommended more movies from the drama, love story and action genre. This is the desired behavior of a RS, i.e. it considers the user behavior instead of merely selecting main-stream choices. The connection to 'Turbo: A Power Ranges Movie', however, remains an open question.

This insight was created to emphasize two critical aspects of recommender system evaluation: the choice of metrics and user-behavior simulator. GRU had the highest accuracy on ML-25M, followed by TPGR, in the offline setting. However, both methods had a poor item coverage as well. After deploying these models to production, as it was simulated here, they do not expand their item coverage but rather leverage popular choices learned from batch. As not everybody likes Star Wars, for example, Wolpertinger outperformed the previously better evaluated methods in the user simulator. As a consequence, a recommender agent should always be investigated further, beyond standard metrics, to ensure that it meets a company's requirements.

6

Discussion & Future Work

The goal of this work was to evaluate several model-free approaches of reinforcement learning based recommender systems in a fair setting. In addition to that, two more recent Reinforcement Learning (RL) algorithms have been tested as well for a possible improvement of existing methods in the recommender domain. This concluding chapter discusses the results of this thesis, emphasizes its limitations and outlines possible future work.

While RL algorithms outperform humans in multiple games and enhance robot control systems, their application in the recommendation domain is subject to multiple challenges. The possibly most restrictive challenge is that Recommender System (RS) need to learn from a recorded batch of data rather than through unlimited interactions with a simulator. Especially the employment of on-policy methods suffers from this because its update procedure enforces that the actions are produced from the current policy. Therefore, it is common to introduce an off-policy compensation to make these approaches possible after all. In this work, Importance Sampling with weight clipping to upper-bound the likelihood was used and the behavior policy was approximated by learning an extra neural network jointly, following [6]. The disadvantage of this approach, however, is that it takes many iterations until the behavior policy is approximated to a certain level so that it is actually helpful to the target policy. Having this model available from the very beginning could drastically decrease time until convergence of the model.

Even though off-policy algorithms are theoretically able to improve the

current policy through samples from another behavior policy, they show problems in practice. One of the challenges is that many state-action combinations never occur in the collected dataset and hence value functions need to extrapolate them. It was shown in related work that this commonly leads to drastic value overestimation [8]. For this reason, two algorithms, Random Ensemble Mixture (REM) [71] and Quantile Regression DQN (QR-DQN) [91], were applied on the recommendation task as they have been shown to enhance performance over the classic 'online' Deep Q-Network (DQN) approach. 'Online' here refers to the ability to directly interact with an environment and hence to allow virtually unlimited trial-and-error interactions. Surprisingly, neither of these algorithms could improve the recommender engine, but they were rather outperformed by other methods. One limitation of their application is that the performance was only evaluated for one parameterization of value-heads and quantiles, $K = 25$, which could have been an unfortunate choice. While they were originally proposed with larger values, e.g. $K = 200$, this is simply not possible in the classic recommendation application, when no other module significantly reduces the full item catalogue to a set of candidates.

The large number of discrete actions is the second main challenge for enhancements of RL agents in this field. Especially the application of value-based approaches is restricted by that as a value needs to be computed for every available action. This work did not collect quantitative metrics that measured the duration of training or inference. The main reason for this is that many computation jobs are necessary to train multiple seeds of several algorithms for three datasets. As those computations are executed on a cluster that contains different hardware specifications, a measured duration would not be comparable. However, especially TPGR and Wolpertinger were trained significantly faster than e.g. NEWS or Dueling.

All of the approaches compared in this work have had varying contributions, such as handling the large action space, predicting a whole list or learning from offline data. This was especially visible in the necessary reduction of the action space. It was shown that some methods are only feasible in smaller action spaces, i.e. if a subset of candidates is provided. However, in the dimension where these borders overlap, e.g. ML-1M, a direct comparison is of high interest in order to select the right model. This number of items is also a reasonable size of a possible candidate set, which emphasizes its importance of comparability. Finally, the difference of requirements for every individual recommender application, e.g. YouTube vs. MovieLens deployment, is one of the main reasons why every publication defines their own evaluation setting. Hence, direct comparability will most likely remain an open issue in this field as it is mainly driven by industry on proprietary

datasets rather than academic researchers.

Generally speaking, there was not one RL-approach that was better than the others after all in this evaluation. Each dataset resulted in a completely different ranking between the approaches that were evaluated in this thesis. Here, the focus was rather on the classification accuracy, measured by the Hitrate, than on additional ranking metrics that emphasize the position of the correct item in a list. This could have shown more granular differences between the approaches. Additionally, gathered diversity metrics focused on repetitions of item entities rather than similarities between products, e.g. given by content representations. Instead, an item was exclusively represented by a contextual vector created by Word2Vec [49]. Using content vectors in addition to context information has been shown to further improve recommendation engines [113] and is left for future work with these approaches. Additionally, all agents only used very basic network architectures, while in the supervised RS research far more complicated networks have been established. It is expected that actors can heavily benefit from these more complex advancements such as attention [55] or Variational Autoencoders (VAEs) [57].

None of the RL approaches has shown comparative performance to a simple supervised recurrent agent, neither in the offline evaluation task, nor during online simulation. This probably goes back to the problem that agents are learned from batch and can not explore the full action set. While the offline evaluation is carried out similar in most publications, it was shown that a clear strategy of comparable online evaluation is needed.
The simulation environment that was built, is far from being a solid user behavior model and hence directly impacts evaluation results. However, as the research community only recently started to propose open-source simulation environments, most authors created their own interpretation of a user model. This finally raises the question if the used simulator was created in that particular manner to make the proposed RL-agent better than available baselines. In many cases, the exact procedure is additionally hidden by not publishing the source code to an open-source platform.
Therefore, the most important future work is in the domain of user behavior simulation for implicit feedback datasets. The better the simulator, the better RL agents can be optimized and evaluated. First approaches have been published but they still have too many levels of freedom that can lead to incomparable results henceforth.

As a final proposal of future work, no approach was found in related work of recommender systems that leveraged countermeasures to overestimation problems such as eligibility traces [66]. Additionally, a recurrent actor was used in this work, and in many others, without further consideration of its

impact to the policy. R2D2 [114], for example, studied the usage of actors with a Recurrent Neural Network (RNN) and found that storing additional information in the replay buffer can further enhance the performance. In general, RL research moves more and more into the direction of distributed distributional approaches. The application of D4PG [115], a distributional Actor-Critic method, sounds promising as the large action space can be handled most efficiently with an actor that reduces the whole item space to a candidate subset while benefiting from the stability of a critic.

At the end of the day, one should weigh the decision to create a RL-based RS up against its variety of pitfalls. Value overestimation problems, a high level of randomness and a large number of hyperparameters are not always worth the potential small benefits to a more simple but elaborated approach.

Conclusion

This thesis studied the lack of comparability of model-free RL-based recommender systems. It was revealed that most papers prohibit direct comparison due to the use of proprietary datasets, different user behavior simulators or a varying evaluation strategy. Therefore, a straight-forward evaluation protocol for interactive recommender systems was devised and applied to several approaches in this field. As an addition, the training procedure for off-policy agents from a collected set of user trajectories is detailed as most papers merely provide vague descriptions. It was found that value-based methods could only effectively learn from batch if additional negative examples were added to the experience buffer in order to reduce value overestimation and stabilize training.

Results indicate that most publications under review do not outperform other baselines on the evaluated datasets as it was claimed by some authors. Further, it was found that the majority of considered RL methods are approximately equally accurate but were surpassed by a popular supervised-learning baseline nevertheless. This finally emphasizes the need of identical and reproducible evaluations and the choice of appropriate and strong baselines. QR-DQN and REM, two recent approaches that have shown good performance in offline RL, were applied for the first time in the recommender domain. REM achieved competitive results to the other methods but could not outperform them significantly. Contrarily, QR-DQN was one of the the worst agents in this evaluation while taking the longest to train.

The large discrete action space remains an open challenge that prohibits the direct application of new advances in the general RL field to this domain. Therefore, new approaches are needed that can combine these enhancements with the restricting properties of the recommendation setting. A next important step towards better comparable papers is to create an open-source user simulator that can be used out-of-the-box. While other domains, such as Natural Language Processing, have typical benchmark datasets to emphasize contributions, there does not yet exist an established similar tool for the research of interactive recommendation methods.

References

[1] McKinsey and Company. *How retailers can keep up with consumers*. 2013. URL: https://www.mckinsey.com/industries/retail/our-insights/how-retailers-can-keep-up-with-consumers (visited on 03/14/2020).

[2] CNET. *YouTube's AI is the puppet master over most of what you watch*. 2018. URL: https://www.cnet.com/news/youtube-ces-2018-neal-mohan/ (visited on 03/14/2020).

[3] Guy Shani, David Heckerman, and Ronen I Brafman. „An MDP-based recommender system". In: *Journal of Machine Learning Research* 6.Sep (2005), pp. 1265–1295.

[4] David Silver et al. „Mastering the game of go without human knowledge". In: *Nature* 550.7676 (2017), pp. 354–359.

[5] Oriol Vinyals et al. „Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *Nature* 575.7782 (2019), pp. 350–354.

[6] Minmin Chen et al. „Top-k off-policy correction for a REINFORCE recommender system". In: *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*. ACM. 2019, pp. 456–464.

[7] Xiangyu Zhao et al. „Deep reinforcement learning for list-wise recommendations". In: *arXiv preprint arXiv:1801.00209* (2017).

[8] Scott Fujimoto, David Meger, and Doina Precup. „Off-policy deep reinforcement learning without exploration". In: *arXiv preprint arXiv:1812.02900* (2018).

[9] Mohit Sewak. *Deep Reinforcement Learning: Frontiers of Artificial Intelligence*. Springer Singapore, Jan. 2019. DOI: 10.1007/978-981-13-8285-7.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018.

[11] Masashi Sugiyama, Hirotaka Hachiya, and Tetsuro Morimura. *Statistical Reinforcement Learning: Modern Machine Learning Approaches*. 1st. Chapman & Hall/CRC, 2013. ISBN: 9781439856895.

[12] Richard Bellman. „Dynamic programming and Lagrange multipliers". In: *Proceedings of the National Academy of Sciences of the United States of America* 42.10 (1956), p. 767.

[13] Timothy P Lillicrap et al. „Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[14] Michel Tokic and Günther Palm. „Value-difference based exploration: adaptive control between epsilon-greedy and softmax". In: *Annual Conference on Artificial Intelligence*. Springer. 2011, pp. 335–346.

[15] Eduardo Rodrigues Gomes and Ryszard Kowalczyk. „Dynamic analysis of multiagent Q-learning with *epsilon*-greedy exploration". In: *Proceedings of the 26th annual international conference on machine learning*. ACM. 2009, pp. 369–376.

[16] Peter Dayan. „The convergence of TD (lambda) for general lambda". In: *Machine learning* 8.3-4 (1992), pp. 341–362.

[17] Christopher John Cornish Hellaby Watkins. „Learning from delayed rewards". In: (1989).

[18] Hado V Hasselt. „Double Q-learning". In: *Advances in Neural Information Processing Systems*. 2010, pp. 2613–2621.

[19] Ronald J Williams. „Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[20] John Schulman et al. „Trust region policy optimization". In: *International conference on machine learning*. 2015, pp. 1889–1897.

[21] Kai Arulkumaran et al. „A brief survey of deep reinforcement learning". In: *arXiv preprint arXiv:1708.05866* (2017).

[22] Richard S Sutton et al. „Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems*. 2000, pp. 1057–1063.

[23] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*. Vol. 1. Citeseer, 2017.

[24] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. „Multilayer feedforward networks are universal approximators“. In: *Neural networks* 2.5 (1989), pp. 359–366.

[25] Vinod Nair and Geoffrey E Hinton. „Rectified linear units improve restricted boltzmann machines“. In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[26] Sepp Hochreiter and Jürgen Schmidhuber. „Long short-term memory“. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[27] Kyunghyun Cho et al. „Learning phrase representations using RNN encoder-decoder for statistical machine translation“. In: *arXiv preprint arXiv:1406.1078* (2014).

[28] Yann LeCun, Yoshua Bengio, et al. „Convolutional networks for images, speech, and time series“. In: *The handbook of brain theory and neural networks* 3361.10 (1995), p. 1995.

[29] Ashish Vaswani et al. „Attention is all you need“. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[30] Trapit Bansal, David Belanger, and Andrew McCallum. „Ask the gru: Multi-task learning for deep text recommendations“. In: *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM. 2016, pp. 107–114.

[31] Balázs Hidasi et al. „Session-based recommendations with recurrent neural networks“. In: *arXiv preprint arXiv:1511.06939* (2015).

[32] Benjamin M Marlin. „Modeling user rating profiles for collaborative filtering“. In: *Advances in neural information processing systems*. 2004, pp. 627–634.

[33] Robin Van Meteren and Maarten Van Someren. „Using content-based filtering for recommendation“. In: *Proceedings of the Machine Learning in the New Information Age: MLnet/ECML2000 Workshop*. Vol. 30. 2000, pp. 47–56.

[34] Jun Wang, Arjen P De Vries, and Marcel JT Reinders. „Unifying user-based and item-based collaborative filtering approaches by similarity fusion“. In: *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. 2006, pp. 501–508.

[35] Francesco Ricci, Lior Rokach, and Bracha Shapira. „Introduction to Recommender Systems Handbook“. In: *Recommender Systems Handbook*. Springer, 2011, pp. 1–35.

[36] Badrul Sarwar et al. „Item-based collaborative filtering recommendation algorithms“. In: *Proceedings of the 10th international conference on World Wide Web*. 2001, pp. 285–295.

[37] Greg Linden, Brent Smith, and Jeremy York. „Amazon. com recommendations: Item-to-item collaborative filtering“. In: *IEEE Internet computing* 7.1 (2003), pp. 76–80.

[38] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. „Are we really making much progress? A worrying analysis of recent neural recommendation approaches“. In: *Proceedings of the 13th ACM Conference on Recommender Systems*. ACM. 2019, pp. 101–109.

[39] Hao Ma et al. „Sorec: social recommendation using probabilistic matrix factorization“. In: *Proceedings of the 17th ACM conference on Information and knowledge management*. 2008, pp. 931–940.

[40] Linas Baltrunas, Bernd Ludwig, and Francesco Ricci. „Matrix factorization techniques for context aware recommendation“. In: *Proceedings of the fifth ACM conference on Recommender systems*. 2011, pp. 301–304.

[41] Robert Bell, Yehuda Koren, and Chris Volinsky. „Modeling relationships at multiple scales to improve accuracy of large recommender systems“. In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2007, pp. 95–104.

[42] Yehuda Koren. „Collaborative filtering with temporal dynamics“. In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2009, pp. 447–456.

[43] Andriy Mnih and Russ R Salakhutdinov. „Probabilistic matrix factorization“. In: *Advances in neural information processing systems*. 2008, pp. 1257–1264.

[44] Dheeraj Bokde, Sheetal Girase, and Debajyoti Mukhopadhyay. „Matrix factorization model in collaborative filtering algorithms: A survey“. In: *Procedia Computer Science* 49 (2015), pp. 136–146.

[45] Andrew Zimdars, David Maxwell Chickering, and Christopher Meek. „Using temporal data for making recommendations“. In: *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc. 2001, pp. 580–588.

[46] Bamshad Mobasher et al. „Using sequential and non-sequential patterns in predictive web usage mining tasks“. In: *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* IEEE. 2002, pp. 669–672.

[47] Duen-Ren Liu, Chin-Hui Lai, and Wang-Jung Lee. „A hybrid of sequential rules and collaborative filtering for product recommendation“. In: *Information Sciences* 179.20 (2009), pp. 3505–3519.

[48] Kaiming He et al. „Deep residual learning for image recognition“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[49] Tomas Mikolov et al. „Efficient estimation of word representations in vector space“. In: *arXiv preprint arXiv:1301.3781* (2013).

[50] Richard Socher et al. „Parsing natural scenes and natural language with recursive neural networks“. In: *Proceedings of the 28th international conference on machine learning (ICML-11).* 2011, pp. 129–136.

[51] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. „Restricted Boltzmann machines for collaborative filtering“. In: *Proceedings of the 24th international conference on Machine learning.* 2007, pp. 791–798.

[52] James Bennett, Stan Lanning, et al. „The netflix prize“. In: *Proceedings of KDD cup and workshop.* Vol. 2007. Citeseer. 2007, p. 35.

[53] Balázs Hidasi and Alexandros Karatzoglou. „Recurrent neural networks with top-k gains for session-based recommendations“. In: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management.* 2018, pp. 843–852.

[54] Chao-Yuan Wu et al. „Recurrent recommender networks“. In: *Proceedings of the tenth ACM international conference on web search and data mining.* 2017, pp. 495–503.

[55] Jing Li et al. „Neural attentive session-based recommendation“. In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management.* 2017, pp. 1419–1428.

[56] Jiaxi Tang and Ke Wang. „Personalized top-n sequential recommendation via convolutional sequence embedding“. In: *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining.* 2018, pp. 565–573.

[57] Dawen Liang et al. „Variational autoencoders for collaborative filtering“. In: *Proceedings of the 2018 World Wide Web Conference.* 2018, pp. 689–698.

[58] Yao Wu et al. „Collaborative denoising auto-encoders for top-n recommender systems“. In: *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. 2016, pp. 153–162.

[59] Wang-Cheng Kang and Julian McAuley. „Self-attentive sequential recommendation“. In: *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE. 2018, pp. 197–206.

[60] Fei Sun et al. „BERT4Rec: Sequential recommendation with bidirectional encoder representations from transformer“. In: *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 2019, pp. 1441–1450.

[61] Shuai Zhang et al. „Deep learning based recommender system: A survey and new perspectives“. In: *ACM Computing Surveys (CSUR)* 52.1 (2019), pp. 1–38.

[62] Joeran Beel et al. „Research paper recommender system evaluation: a quantitative literature survey“. In: *Proceedings of the International Workshop on Reproducibility and Replication in Recommender Systems Evaluation*. 2013, pp. 15–22.

[63] Malte Ludewig et al. „Performance comparison of neural and non-neural approaches to session-based recommendation“. In: *Proceedings of the 13th ACM Conference on Recommender Systems*. 2019, pp. 462–466.

[64] Volodymyr Mnih et al. „Playing atari with deep reinforcement learning“. In: *arXiv preprint arXiv:1312.5602* (2013).

[65] Volodymyr Mnih et al. „Human-level control through deep reinforcement learning“. In: *Nature* 518.7540 (2015), p. 529.

[66] Matteo Hessel et al. „Rainbow: Combining improvements in deep reinforcement learning“. In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[67] Tom Schaul et al. „Prioritized experience replay“. In: *arXiv preprint arXiv:1511.05952* (2015).

[68] Scott Fujimoto, Herke Hoof, and David Meger. „Addressing Function Approximation Error in Actor-Critic Methods“. In: *International Conference on Machine Learning*. 2018, pp. 1587–1596.

[69] Tuomas Haarnoja et al. „Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor“. In: *arXiv preprint arXiv:1801.01290* (2018).

[70] Sascha Lange, Thomas Gabel, and Martin Riedmiller. „Batch reinforcement learning“. In: *Reinforcement learning*. Springer, 2012, pp. 45–73.

[71] Rishabh Agarwal, Dale Schuurmans, and Mohammad Norouzi. „An Optimistic Perspective on Offline Reinforcement Learning“. In: *NeurIPS Deep Reinforcement Learning Workshop*. Contributed Talk at NeurIPS 2019 DRL Workshop. 2019. URL: `https://arxiv.org/abs/1907.04543`.

[72] Scott Fujimoto et al. „Benchmarking Batch Deep Reinforcement Learning Algorithms“. In: *arXiv preprint arXiv:1910.01708* (2019).

[73] Xiangyu Zhao et al. „Deep reinforcement learning for search, recommendation, and online advertising: a survey“. In: *ACM SIGWEB Newsletter* Spring (2019), pp. 1–15.

[74] Haokun Chen et al. „Large-scale interactive recommendation with tree-structured policy gradient“. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 3312–3320.

[75] Gabriel Dulac-Arnold et al. „Deep reinforcement learning in large discrete action spaces“. In: *arXiv preprint arXiv:1512.07679* (2015).

[76] Ting Liu et al. „An investigation of practical approximate nearest neighbor algorithms“. In: *Advances in neural information processing systems*. 2005, pp. 825–832.

[77] Peter Sunehag et al. „Deep reinforcement learning with attention for slate markov decision processes with high-dimensional states and actions“. In: *arXiv preprint arXiv:1512.01124* (2015).

[78] Xiangyu Zhao et al. „Deep reinforcement learning for page-wise recommendations“. In: *Proceedings of the 12th ACM Conference on Recommender Systems*. ACM. 2018, pp. 95–103.

[79] Li Xu et al. „Deep convolutional neural network for image deconvolution“. In: *Advances in neural information processing systems*. 2014, pp. 1790–1798.

[80] Feng Liu et al. „Deep reinforcement learning based recommendation with explicit user-item interactions modeling“. In: *arXiv preprint arXiv:1810.12027* (2018).

[81] Xiangyu Zhao et al. „Model-Based Reinforcement Learning for Whole-Chain Recommendations“. In: *arXiv preprint arXiv:1902.03987* (2019).

[82] Shi-Yong Chen et al. „Stablizing Reinforcement Learning in Dynamic Environment with Application to Online Recommendation“. In: May 2018.

[83] Xiangyu Zhao et al. „Recommendations with negative feedback via pairwise deep reinforcement learning“. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM. 2018, pp. 1040–1048.

[84] Guanjie Zheng et al. „DRN: A deep reinforcement learning framework for news recommendation“. In: *Proceedings of the 2018 World Wide Web Conference*. International World Wide Web Conferences Steering Committee. 2018, pp. 167–176.

[85] Ziyu Wang et al. „Dueling network architectures for deep reinforcement learning“. In: *arXiv preprint arXiv:1511.06581* (2015).

[86] Eugene Ie et al. „SlateQ: A tractable decomposition for reinforcement learning with recommendation sets“. In: (2019).

[87] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. „Challenges of real-world reinforcement learning“. In: *arXiv preprint arXiv:1904.12901* (2019).

[88] Nan Jiang and Lihong Li. „Doubly robust off-policy value evaluation for reinforcement learning“. In: *arXiv preprint arXiv:1511.03722* (2015).

[89] Olivier Pietquin et al. „Sample-efficient batch reinforcement learning for dialogue management optimization“. In: *ACM Transactions on Speech and Language Processing (TSLP)* 7.3 (2011), pp. 1–21.

[90] Yash Chandak et al. „Learning action representations for reinforcement learning“. In: *arXiv preprint arXiv:1902.00183* (2019).

[91] Will Dabney et al. „Distributional reinforcement learning with quantile regression“. In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[92] Hado Van Hasselt, Arthur Guez, and David Silver. „Deep reinforcement learning with double q-learning“. In: *Thirtieth AAAI conference on artificial intelligence*. 2016.

[93] Carlos A Gomez-Uribe and Neil Hunt. „The netflix recommender system: Algorithms, business value, and innovation“. In: *ACM Transactions on Management Information Systems (TMIS)* 6.4 (2015), pp. 1–19.

[94] Tobias Schnabel et al. „Short-term satisfaction and long-term coverage: Understanding how users tolerate algorithmic exploration“. In: *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. 2018, pp. 513–521.

[95] Lixin Zou et al. „Pseudo Dyna-Q: A Reinforcement Learning Framework for Interactive Recommendation“. In: *Proceedings of the 13th International Conference on Web Search and Data Mining*. 2020, pp. 816–824.

[96] F Maxwell Harper and Joseph A Konstan. „The movielens datasets: History and context“. In: *Acm transactions on interactive intelligent systems (tiis)* 5.4 (2016), p. 19.

[97] Ayush Singhal, Pradeep Sinha, and Rakesh Pant. „Use of deep learning in modern recommendation system: A summary of recent works“. In: *arXiv preprint arXiv:1712.07525* (2017).

[98] Steffen Rendle. *Context-aware ranking with factorization models*. Springer, 2011.

[99] Radim Rehurek and Petr Sojka. „Software framework for topic modelling with large corpora“. In: *In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Citeseer. 2010.

[100] David Rohde et al. „Recogym: A reinforcement learning environment for the problem of product recommendation in online advertising“. In: *arXiv preprint arXiv:1808.00720* (2018).

[101] Eugene Ie et al. „RecSim: A Configurable Simulation Platform for Recommender Systems“. In: *arXiv preprint arXiv:1909.04847* (2019).

[102] Steffen Rendle et al. „BPR: Bayesian personalized ranking from implicit feedback“. In: *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*. 2009, pp. 452–461.

[103] Yehuda Koren, Robert Bell, and Chris Volinsky. „Matrix factorization techniques for recommender systems“. In: *Computer* 42.8 (2009), pp. 30–37.

[104] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[105] Maciej Kula. *Spotlight*. `https://github.com/maciejkula/spotlight`. 2017.

[106] Gunnar Schröder, Maik Thiele, and Wolfgang Lehner. „Setting goals and choosing metrics for recommender system evaluations“. In: *UCERSTI2 Workshop at the 5th ACM conference on recommender systems, Chicago, USA*. Vol. 23. 2011, p. 53.

[107] Malte Ludewig and Dietmar Jannach. „Evaluation of session-based recommendation algorithms“. In: *User Modeling and User-Adapted Interaction* 28.4-5 (2018), pp. 331–390.

[108] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge university press, 2008.

[109] Wouter Kool, Herke Van Hoof, and Max Welling. „Stochastic Beams and Where To Find Them: The Gumbel-Top-k Trick for Sampling Sequences Without Replacement“. In: *International Conference on Machine Learning*. 2019, pp. 3499–3508.

[110] Massimo Quadrana, Paolo Cremonesi, and Dietmar Jannach. „Sequence-aware recommender systems“. In: *ACM Computing Surveys (CSUR)* 51.4 (2018), pp. 1–36.

[111] Diederik P Kingma and Jimmy Ba. „Adam: A method for stochastic optimization“. In: *arXiv preprint arXiv:1412.6980* (2014).

[112] Zafarali Ahmed et al. „Understanding the Impact of Entropy on Policy Optimization“. In: *International Conference on Machine Learning*. 2019, pp. 151–160.

[113] Christina Christakou, Spyros Vrettos, and Andreas Stafylopatis. „A hybrid movie recommender system based on neural networks“. In: *International Journal on Artificial Intelligence Tools* 16.05 (2007), pp. 771–792.

[114] Steven Kapturowski et al. „Recurrent experience replay in distributed reinforcement learning“. In: (2018).

[115] Gabriel Barth-Maron et al. „Distributed distributional deterministic policy gradients“. In: *arXiv preprint arXiv:1804.08617* (2018).

Glossary

CDF Cumulative Distribution Function. 39, 40

CF Collaborative Filtering. 23, 24, 25, 49, 51, 56

CNN Convolutional Neural Network. 21, 25

CTR Click Through Rate. 55

DDPG Deep Deterministic Policy Gradient. 26, 27, 37, 39

DP Dynamic Programming. 9, 14, 15

DQN Deep Q-Network. 3, 26, 39, 40, 41, 76

DRL Deep Reinforcement Learning. 2, 3, 4, 25, 26, 27, 39

GPI Generalized Policy Iteration. 12, 15

GPU Graphics Processing Unit. 38

GRU Gated Recurrent Unit. 21, 41, 56, 57

IS Importance Sampling. 33, 36, 59, 75

kNN k-nearest neighbor. 25, 27, 28, 37, 38, 56

LSTM Long Short-Term Memory. 25

MC Monte Carlo. 14, 15

MDP Markov Decision Process. 2, 3, 4, 5, 6, 7, 11, 14, 15, 16, 18, 26, 31, 36, 57

MF Matrix Factorization. 24

ML MovieLens. 48

MLP Multilayer Perceptron. 20, 21, 36

MSE mean squared error. 43, 44, 66

NLP Natural Language Processing. 25, 79

PCA Principal Component Analysis. 35

PG Policy Gradient. 16, 17

QR-DQN Quantile Regression DQN. 39, 40, 41, 57, 59, 66, 69, 76

REM Random Ensemble Mixture. 40, 57, 59, 66, 69, 72, 76

RL Reinforcement Learning. 2, 3, 5, 8, 12, 15, 16, 26, 27, 33, 34, 36, 39, 40, 42, 45, 56, 59, 66, 70, 75, 76, 77, 78, 79

RNN Recurrent Neural Network. 21, 22, 25, 41, 78

RS Recommender System. 2, 3, 26, 28, 32, 34, 36, 42, 44, 45, 53, 54, 55, 74, 75, 77, 78

SVD Singular Value Decomposition. 24

TD Temporal-Difference. 14, 15, 19

TD-Error Temporal-Difference Error. 14, 16, 19, 20, 26, 37, 40, 43, 61, 62

VAE Variational Autoencoder. 25, 77