TU WIEN Informatics

A Comparative Performance Analysis of Deep Reinforcement Learning News Recommender Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Data Science

eingereicht von

Dominik Veselý, BSc.
Matrikelnummer 01633647

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dr.techn. Peter Knees

Wien, 29. September 2023

Dominik Veselý	Peter Knees

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

A Comparative Performance Analysis of Deep Reinforcement Learning News Recommender Systems

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Data Science

by

Dominik Veselý, BSc.
Registration Number 01633647

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dr.techn. Peter Knees

Vienna, 29th September, 2023

Dominik Veselý	Peter Knees

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Dominik Veselý, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. September 2023

Dominik Veselý

Danksagung

Danke an meine Eltern, für ihre Unterstützung während meines Studiums. Danke an Lena, für die emotionale Hilfe während dem Verfassen dieser Arbeit. Danke an all jene, die nie aufgehört haben mich zu fragen, wann ich fertig sein werde.

Acknowledgements

Thanks to my parents, for their support during my studies. Thanks to Lena, for the emotional assistance during the work on this thesis. Thanks to those who never stopped asking me when I'll be done.

What we do in life,
echoes in eternity.

Kurzfassung

Mit der zunehmenden Nutzung von Online-Nachrichtendiensten, sowie dem Überfluss von Nachrichteninhalten, wächst der Bedarf an News Recommender Systems, die Nutzer*innen ihren Interessen entsprechende Nachrichtenbeiträge anbieten. Obwohl es sich nach wie vor um einen unterrepräsentierten Bereich in der Forschung zu Recommender Systemen handelt, so hat das Thema Nachrichtenempfehlung in den letzten Jahren einen Aufschwung erlebt. Vor allem die Veröffentlichung des frei zugänglichen, großen Datensatzes im Jahr 2020, namens "MIND", sowie der damit verbundene Wettbewerb, haben die Forschung weiter vorangetrieben.

Ein weiteres Thema, das einen Anstieg an Forschungsaktivitäten verzeichnen kann, ist Deep Reinforcement Learning, seit der Veröffentlichung bahnbrechender, wissenschaftlicher Arbeiten zu Deep Q-Networks im Jahr 2013 und 2015. Die prominentesten Erfolge des Paradigmas liegen im Bereich der (Video-)Spiele. Nichtsdestotrotz wurde es auch auf eine Vielzahl anderer Probleme angewendet, einschließlich von Recommender Systems.

Die folgende Arbeit verknüpft diese beiden Themen. Nachdem wir das notwendige Hintergrundwissen bereitstellen, stellen wir den aktuellen Stand der Forschung in den Bereichen News Recommender Systems, sowie Deep Reinforcement Learning Recommender Systems vor. Anschließend präsentieren wir den MIND Datensatz. Schließlich stellen wir das Reinforcement Learning Framework vor, das wir rund um diesen Datensatz entwickelt haben. Basierend darauf führen wir eine umfassende komparative Leistungsanalyse von Deep Reinforcement Learning News Recommender Systems durch. Neben dem Testen verschiedener Algorithmen vergleichen wir auch diverse News- und User-Encoder. Unsere reproduzierbaren Experimente decken eine Vielzahl von Ansätzen ab und bieten Einblicke darüber, welche Algorithmen und Encoder am besten für das Problem der Nachrichtenempfehlung geeignet sind. Die Ergebnisse des MIND Wettbewerbs für einen dieser Ansätze, auch wenn diese nicht bahnbrechend sind, unterstreichen die grundsätzliche Eignung und Anwendbarkeit von Deep Reinforcement Learning News Recommender Systems.

Zusammenfassend adressiert diese Arbeit den Mangel an einer Open-Source, reproduzierbaren und umfassenden komparativen Leistungsanalyse. Wir sind der Meinung, dass diese Arbeit und der veröffentlichte Code, im Hinblick auf das bereitgestellte Hintergrundwissen und die Ergebnisse der Experimente, als Ausgangspunkt für weitere Forschung dienen kann.

Abstract

With an increasing amount of news consumption taking place online, and an abundance of news content, comes the need for news recommender systems to serve users with news items that match their interests. While being an underrepresented domain among recommender system research, news recommenders have experienced a surge in the past decade. Especially the 2020 release of a public, large-scale news recommendation dataset and its accompanying competition, called "MIND", has further encouraged research, due to the previous lack of access to comparable data.

Another topic that has seen a rise in research activity is reinforcement learning, particularly deep reinforcement learning, since the release of seminal papers on deep Q-networks in 2013 and 2015. While the learning paradigm's most prominent achievements come from the area of (video) games, reinforcement learning has been applied to various other problems, including the recommendation problem.

This thesis conjoins these two topics. After providing the fundamental background knowledge required to understand the thesis, we present the state-of-the-art in news recommender systems, as well as deep reinforcement learning recommender systems. Subsequently, we discuss and critique the MIND dataset. Finally, we put forward the reinforcement learning framework that we have developed around it. We use this framework to conduct a comprehensive and reproducible comparative performance analysis of deep reinforcement learning news recommender systems. Along with testing several different algorithms, we also compare and contrast different news and user encoders. Our experiments cover a multitude of approaches, offering insights into which algorithms and encoders are most suitable for the news recommendation problem. The MIND competition results of one of these approaches, albeit not achieving ground-breaking scores, underline the general viability of deep reinforcement learning news recommender systems.

In summary, this work addresses the lack of an open-source, reproducible and broad comparative performance analysis. We believe that this thesis, both in terms of the provided theoretical background and the experiment results, along with the published code, can serve as a starting point for further research.

Contents

CHAPTER 1

Introduction

In this chapter, we begin by introducing the topic of this thesis. We will first discuss the motivation for this work, and subsequently outline the problem statement and the aim of this work. Finally, we will address the general structure of this thesis.

1.1 Motivation and Problem Statement

This thesis conjoins two topics that have experienced an increased amount of research in the past decade, namely news recommender systems and deep reinforcement learning.

In this day and age, an ever-increasing proportion of news consumption takes place online [14]. Readers rely on news websites, search engines, and news aggregators, e.g. Google News, Microsoft News, Flipboard, etc., to stay informed. This shift entails that publishers are not confined to the limited amount of space offered by traditional news media, such as print, TV and radio, but are free to utilize the essentially infinite storage space of the world wide web. The consequence is an abundance of news content, leading to information overload that ultimately burdens the consumer. With this comes a need, and a possibility, for news recommender systems, in order to alleviate users from the aforementioned pressure, and keep them up-to-date with news item suggestions, tailored to their personal interests. In comparison to other recommendation problem domains, the news domain poses a set of unique challenges that are not present, or not as strongly pronounced in other domains. On the supply side, the news publishing environment is highly dynamic, with an enormous amount of content being released at every hour of every day. At the same time, the relevancy of this content is typically of short duration, with most information being irrelevant within days. On the demand side, user's interests can evolve rapidly, e.g. due to current events, or depending on locality. Therefore, a recommender system must be able to distinguish between long- and short-term interests of users and react accordingly. Furthermore, albeit this is beyond the scope of this work, news recommenders are arguably more important than other

recommender systems from a societal perspective. A broadly and diversely informed citizenry that engages with counter-attitudinal information is paramount for democratic processes and institutions. To that end, evaluation methods that go beyond accuracy must be explored. Nonetheless, the specific domain of news recommendation has been underrepresented in recommender systems research, in comparison to other domains, such as movie or product recommendations.

The second overarching topic of this thesis is reinforcement learning, which has been successfully employed in the past decade to solve various problems. The presumably most prominent achievements come from the area of (video) games. Reinforcement learning is a computational approach to the idea of learning from interactions. A reinforcement learning agent aims to learn appropriate mappings from observed states of the environment it interacts with, to actions it can take that will help the agent reach its ultimate goal.

In recent years, research output on the application of reinforcement learning techniques to recommendation problems has increased significantly. However, studies revolving in the news domain are exceedingly rare and usually conducted on proprietary datasets. To that end, the 2020 release of a large-scale, public news recommendation dataset by Microsoft greatly facilitates research in this area, with the accompanying competition allowing a direct comparison with other solutions. However, at the time of writing, deep reinforcement learning based recommenders have not been trained or tested on the dataset. In summary, there currently is a shortage of open-source, reproducible research on the application of deep reinforcement learning algorithms to the news recommendation problem, using a publicly available dataset. Furthermore, there is a complete lack of studies comparing the results achieved by different reinforcement learning algorithms. Therefore, this work aims to fill this space.

1.2 Aim of the Work

The aim of this work is to apply multiple deep reinforcement learning algorithms to the news recommendation problem, and compare and analyze their performance. Furthermore, we will test and compare different news and user encoding approaches. To this end, the aforementioned dataset by Microsoft, i.e. the MIND dataset [93], will serve as the source of data for this project and will thus be used for the training and evaluation of agents. The public MIND competition results will be used to compare our results with recommender systems based on other approaches. Ultimately, this constitutes the implementation and subsequent comparative analysis of a broad spectrum of deep reinforcement learning news recommenders, as well as news and user encoding approaches. To us, reproducibility and open-sourced code is of utmost importance. The source code for the practical part of this thesis can be accessed via GitHub[1]. Finally, the following research questions will be answered throughout the thesis:

[1] https://github.com/d-vesely/drlnrs

- **Q1**: How does the news recommendation problem fit into the reinforcement learning framework?
- **Q2**: How can offline behavior data be used to train a reinforcement learning agent?
- **Q3**: In terms of performance, how do deep reinforcement learning news recommender systems compare to each other, depending on the used algorithms?
- **Q4**: In terms of performance, how do deep reinforcement learning news recommender systems compare to each other, depending on the used news/user encoder?
- **Q5**: In terms of performance, how do deep reinforcement learning news recommender systems compare to others based on different methods?
- **Q6**: How suitable is the application of deep reinforcement learning for recommendation in the news domain?

1.3 Structure of the Work

We will begin with a self-contained explanation of the background knowledge that is required for further reading and understanding of this thesis. Our goal is to provide sufficient information, such that the contents of this work can be understood by a fellow Data Science student that has not taken any elective courses on these topics, or has studied a different branch of informatics. In that regard, there are three topics that are fundamental: reinforcement learning, recommender systems and deep reinforcement learning. We will introduce each one in the Chapters 2, 3, and 4 respectively. Then, in Chapter 5, we will first introduce the issues and challenges that differentiate recommender systems in the news domain from others. Then, we provide an overview of the current state-of-the-art in news recommender systems, as well as the application of deep reinforcement learning methods to the recommendation problem. Chapter 6 presents the primary dataset used in this thesis. We will summarize the results of our exploratory analysis, as well as explain the conducted preprocessing. Finally, Chapter 7 details the practical part of this thesis. We first explain how the available data was embedded into a reinforcement learning framework, then we discuss how we applied deep reinforcement learning algorithms in detail, and our experiment setup. The results of our comparative analysis are visualized and discussed at the end of the chapter. A summarization and conclusion of the thesis, as well as suggestions for future work, are provided in the final Chapter 8.

The work is written to logically progress from background knowledge to the conducted analysis, with frequent references to previous chapters and sections. Furthermore, acronyms are introduced gradually. Therefore, we suggest reading the text in sequential order, especially to readers who feel they lack sufficient background knowledge.

CHAPTER 2

Fundamentals I: Reinforcement Learning

This chapter covers the fundamentals of Reinforcement Learning (RL). These concepts are part of the groundwork underpinning this thesis, and their understanding is essential for further reading. Part I of the standard work on reinforcement learning, by Sutton and Barto [83], is the main source for this chapter. Additional information was sourced from the chapters 1, 2, 17, 18 and 21 of the standard work on artificial intelligence, by Russell and Norvig [73].

2.1 Relationship to other Paradigms

Before delving into what reinforcement learning actually is, we first describe where it fits into the overarching field of Artificial Intelligence (AI), as well as how it relates to other paradigms of the field. This ensures a shared vocabulary and a correct categorization of the concept. Figure 2.1, shown at the end of this section, summarizes these relationships in the form of a Venn diagram.

2.1.1 Artificial Intelligence

The term "artificial intelligence" was first used by John McCarthy in 1955, in the proposal for a workshop on the topic at Dartmouth College [55]. His theory was that it is possible to describe any feature of intelligence precisely enough for a machine to simulate it. It is difficult to define AI, perhaps because it is just as difficult to define human intelligence. The standard work on the subject, by Russell and Norvig, presents four approaches to defining AI, with the authors arguing for the "rational agent approach" [73]. An agent, i.e. an entity that acts ("agent" originates from the latin word "agēns", the present active participle of "agere", meaning "do" [42]), is in this context a computer program.

It is said to be rational, when it takes actions in accordance with the goal of reaching the best (expected) outcome. The authors even argue that "computational rationality" would have been a more precise terminology than AI for what McCarthy was describing. Nevertheless, AI went on to become the umbrella term for a vast array of subdomains, such as logic, reasoning, search, computer vision, natural language processing, machine learning and more, with many of these fields overlapping to some degree.

2.1.2 Machine Learning

Machine Learning (ML), a subdomain of AI, concerns itself with the ability of rational agents to improve their performance on future tasks through experience and the observation of data, i.e. the agents' capacity to learn. Russell and Norvig identify three main reasons for why it is desirable for an agent to improve on its own, as opposed to being improved by its human creator (usually meaning programmer) [73]:

1. The human creator usually cannot predict all conceivable scenarios involving the agent. For example, game-playing agents have to learn to operate in unseen states of the game, and it is often intractable for the human creator to define the agent's behavior for the entire state-space.

2. The human creator cannot predict changes that occur over time that affect the agent. Again, game-playing agents must be able to adapt to conceptual changes of the game itself, which is common in modern competitive computer games.

3. The human creator might know how to perform a task, but be unable to program or even explain their own approach. For example, humans are able to distinguish whether a seen object is close or far away, but would struggle to program a computer to do the same without applying ML algorithms.

The term "machine learning" was first coined by the then IBM employee Arthur Samuel [75], who was one of 10 attendees of McCarthy's workshop at Dartmouth in 1956 [73], during his work on agents learning to play checkers. Note that ML encompasses a problem, a class of solution methods for said problem, as well as the research field that revolves around both. While there are various forms of ML, the three arguably main types can be distinguished via the kind of feedback available to the agent to learn from.

In "supervised learning" the agent observes labeled data, i.e. inputs with explicit output feedback, and learns a function that generates the output from the input. The domain can be further separated into regression and classification, where problems with continuous outputs fall into the former, and discrete outputs into the latter category. The agent's goal is to maximize its performance, which can be measured in different ways. A simple example of Supervised Learning (SL) is an agent that learns to detect spam e-mails, by observing examples labeled by a supervisor (e.g. a human). In this case, the agent's performance should be measured with precision, since relevant e-mails being marked as spam is less desirable than spam remaining undetected.

In "unsupervised learning" the agent observes unlabeled data, i.e. inputs without explicit output feedback, and learns hidden structures in the input. A typical Unsupervised Learning (UL) problem is clustering, where the agent categorizes the input data into two or more groups, based on the presence or absence of patterns it discovers in the data.

In reality, the lines between supervised and unsupervised learning can be blurred, mainly for two reasons. Firstly, it is possible to use both in conjunction, by supplying the agent with a (usually small) set of labeled data, in order to aid the learning process applied to a (usually large) set of unlabeled data. Secondly, labels can be incorrect, due to random noise or even systematic errors. Therefore, these cases are referred to as "semi-supervised learning".

In "reinforcement learning" the agent receives feedback in the form of reinforcements, which are supposed to reinforce the agent's behavior. They can either be rewards or punishments, depending on whether the agent performed well or not. However, the agent has to learn on its own, which of the agent's actions prior to receiving the reinforcement contributed to it and in what way. Typically, the term "reward" is used for both forms of reinforcement, since rewards are usually numeric and can be both positive (reward) and negative (punishment).

Of course, all of these types of machine learning can be employed in conjunction, as part of a larger system, or as subsystems of each other.

Before going into the details of RL, we briefly want to establish a common understanding of Deep Learning (DL).

2.1.3 Deep Learning

A type of computing structure that is often used for ML is the Artificial Neural Network (ANN). They are inspired by the biological brain and consist of nodes, also called neurons, connected via directed links. Typically, the nodes are arranged in layers. Nodes that are neither in the input nor the output layer are said to be part of a hidden layer. Warren McCulloch and Walter Pitts, the creators of the neural network model, argued in their original paper released in 1943 [56] that arbitrarily deep, and possibly recurrent, networks can achieve any functionality, as long as the number of nodes is large enough. This was later proven with varying assumptions, such as arbitrary depth (number of hidden layers) or arbitrary width (number of units) of the network, leading to multiple so-called universal approximation theorems [43] [32]. Research on neural networks continued throughout the 20th century, leading to important advances in the field. Nevertheless, their popularity waned due to inflated expectations and the improvement of other ML techniques. In the early 2000s, the prevailing sentiment was that training deep neural networks, i.e. networks consisting of two or more hidden layers, was a very difficult undertaking. However, the growing computational capabilities of the 21st century allowed for more experimentation and research. The efforts were spearheaded by the ML research groups headed by Geoffrey Hinton, Yann LeCun, and Yoshua Bengio, leading to the revelation that existing training algorithms did indeed work well in practice [41] [9]

[67]. What followed was a new wave of research on increasingly deep and complex ANNs, causing the popularization of the term "deep learning". These Large Deep Neural Network (LDNN) began outperforming other ML approaches across multiple domains and continued to do so, ensuring the popularity of deep learning up until now, where it has become an umbrella term for a wide variety of techniques and architectures based on LDNNs. Some of them include convolutional neural networks, recurrent neural networks, long short-term memory and transformers [30]. Transformers are based on the attention mechanism and are particularly useful in natural language processing, which we utilize in the practical part of this thesis. The transformer was introduced in the seminal work "Attention Is All You Need" [86].

DL methods can be employed in all three main types of ML presented in the previous section, where the application of DL in RL is commonly referred to as Deep Reinforcement Learning (DRL). The following Venn diagram visually explains the categorizations described in this section.

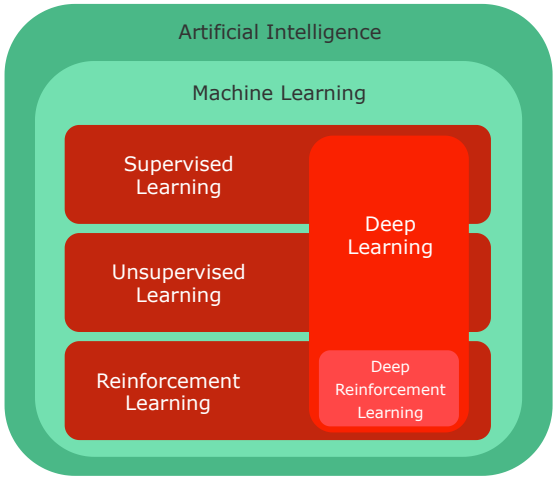


Figure 2.1: AI is an umbrella term, with ML as a subdomain. SL, UL and RL are three main types of ML. DL refers to a broad family of techniques based on LDNNs that can be applied to all types of ML. DRL is the combination of DL and RL.

2.2 Introduction to Reinforcement Learning

The process of learning via interaction with an environment is a fundamental concept that underpins most theories of learning and intelligence in general. It is a very natural concept that humans themselves apply often throughout life. We sense our environment and its responses to our actions, and continually adapt our behavior depending on the received feedback and our goals. Ultimately, we are able to distinguish good from bad behavior with respect to our desired outcome. For example, infants are not necessarily explicitly taught how to crawl, walk, or communicate non-verbally. Instead they act and observe whether their intended response occurs, e.g. their parent hands them a piece of

fruit after the infant gestures for it, otherwise they adapt. The same process occurs when learning a new game or sport, learning to cook, or learning to navigate social interactions.

RL is a computational approach to this concept that, at its core, revolves around finding appropriate mappings from observed states of the environment to actions. This mapping is not taught, but learned, meaning that it has to be inferred from experiences. This is the key difference to SL, where one would present input-output pairs, in this case state-action pairs denoting desired behavior, to the agent to learn from. However, such an approach is often inadequate for interactive learning problems, due to the impracticality of collecting a sufficiently large set of examples that covers a representative sample of all possible situations. This becomes clear when one tries to play the role of the supervisor. While teaching an agent (say person) how to play football, how to cook, or how to drive a car requires guidance from the supervisor in the form of knowledge and feedback, the agent's experience plays a vital role in the learning process that cannot be replaced. A driving instructor would not expect their pupil to learn how to drive solely based on a set of situation-action pairs presented by the instructor, as the construction of such a set is simply infeasible. In contrast, teaching an agent (say person) how to distinguish cats and dogs can be done entirely with a set of examples, from which the agent can derive its own set of rules and generalize, i.e. extrapolate, to previously unseen instances. The absence of labeled data is common in both RL and UL. The key difference there is that UL seeks to find hidden structures in the input data, while the goal of RL is to maximize the sum of received rewards, i.e. the total reward. Generally, RL is significantly more focused on goal-directed learning than the other two types of ML. As already mentioned in the previous section, an RL agent can involve SL to solve a subproblem critical to the complete goal-seeking agent. For example, the detection of traffic signs in a camera feed is a subproblem for an agent that is learning how to drive a car. That being said, modern, practical approaches to RL, which we will discuss in Chapter 4, rely on SL to train certain components of the agent.

Aside from the agent itself, and the environment it continually interacts with, an RL setting consists of three additional main elements:

1. *Reward Signal*: As already discussed, the agent periodically receives a numeric reward directly from the environment. Via its sign and absolute value, the reward defines whether the agent's actions are good or bad, as well as their significance. Since the agent aims to maximize the total reward during its interaction, the reward signal ultimately designates the goal of the particular RL problem. Rewards can be both deterministic and stochastic.

2. *Value Function*: Given a state, and possibly a chosen action, the value function basically returns an estimate of the total reward the agent will collect from that point onward. Therefore, while rewards convey information about the current state and/or action, values are indicators for the desirability of the future, based on the states that follow and the rewards they yield. However, this information is usually only an estimate that has to be continually adapted, following the agent's

experiences. Rewards and values are thus tightly coupled, because the former informs and shapes the latter. Methods for the efficient and precise estimation of values are at the core of RL. The relationship between rewards and values is, again, very natural. Stubbing a toe is followed by a negative reward in the form of pain, but will typically not ruin an entire day. Conversely, eating junk food might be followed by pleasure, but could have long-term impacts on health.

3. *Policy*: The aforementioned mapping from environment states to actions is referred to as the agent's policy. This is a general construct that can take on various forms, depending on the characteristics of the RL problem, ranging from simple lookup tables to complex functions. An agent's policy is closely coupled to value estimations, because an agent generally chooses its actions so as to enter positions of high value. Policies can be both deterministic and stochastic.

In addition to these three components, an RL system can include a model of the environment, which gives the agent the possibility to anticipate how the environment will respond to its actions. The agent can thus employ a technique called planning, which is the process of deciding on a strategy a priori. RL solution methods can be divided into model-free and model-based approaches. Because models of the environment are usually not easy to obtain, the arguably most famous RL algorithms fall into the former category. We are only concerned with model-free methods in this work.

The formal framework for the concepts discussed in this section, that is the bedrock of the RL problem, is the Markov Decision Process (MDP).

2.3 Markov Decision Process

A Markov decision process is a mathematical formalization of the sequential decision problem that occurs in RL. The MDP that frames the interaction between agent and environment, described in the previous section, is formalized as follows. The interaction is split into discrete timesteps, $t = 0, 1, 2, 3, \dots$, at which the agent selects an action based on its observation of the environment's state. The chosen action in turn affects the environment, prompting it to send a reward to the agent and changing its state accordingly. The state, action and reward at each timestep t are denoted as random variables S_t, A_t and R_{t+1}. Note that R_{t+1} denotes the reward due to action A_t, in order to signify that the reward and the next state S_{t+1} are jointly determined by the taken action. Figure 2.2 visualizes this MDP. The interaction between agent and environment thus produces a sequence of states, actions and rewards in the form of $\{S_0, A_0, R_1, S_1, A_1, R_2, \dots\}$, referred to as a trajectory τ.

The sample spaces for the state, action and reward random variables are $\mathcal{S}$, $\mathcal{A}(s)$ and $\mathcal{R} \subset \mathbb{R}$ respectively, where the set of actions can depend on the current state s, and rewards are numeric. If these sets are finite, a probability can be assigned to the occurrence of a particular next state $s' \in \mathcal{S}$ and reward $r \in \mathcal{R}$ at time t, depending on

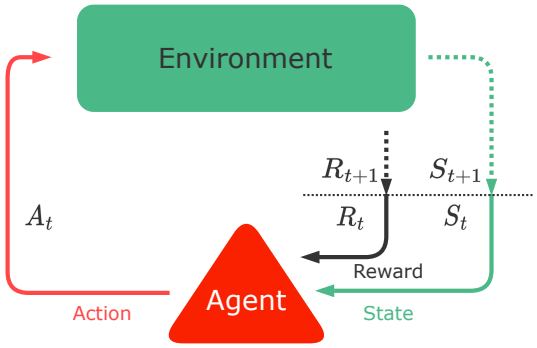


Figure 2.2: The interaction between agent and environment, formalized as an MDP. The agent observes the current state, receives a reward and chooses an action, which affects the environment's state and yields a new reward.

the particular current state $s \in \mathcal{S}$ and the taken action $a \in \mathcal{A}(s)$. These four arguments are mapped to probabilities with a deterministic function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$, defined as:

$$p(s', r \mid s, a) \doteq \mathbb{P}(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a) \tag{2.1}$$

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1 \quad \forall s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{2.2}$$

Equation 2.2 signifies that the probabilities must sum to 1 for each state-action pair, since p defines a probability distribution for each such pair. The states in an MDP fulfill the so-called Markov property, which stipulates that the probabilities for each combination of next state and reward only depend on the preceding state and action, and are entirely independent of any earlier values in the trajectory. Each state representation must thus encompass all relevant information from the past interaction trajectory. Consequently, p represents a complete description of the environment's dynamics.

As already discussed informally in the previous section, the agent seeks to maximize the total cumulative reward it receives in the course of its trajectory. In order to formalize this, we define the return G_t, which the agent tries to maximize in expectation.

$$G_t \doteq R_{t+1} + R_{t+2} + \cdots + R_T \tag{2.3}$$

T designates a final timestep, where we can distinguish finite ($T < \infty$) and infinite horizons ($T = \infty$). Interactions of the former kind are also called episodic tasks, in which the agent eventually reaches a terminal state and the interaction either ends or is reset. Otherwise, we speak of continuing tasks. Simple examples for the two would be an agent playing chess, in contrast to an agent regulating some process in a chemical plant indefinitely. Due to the assumption that $\mathcal{R} \subset \mathbb{R}$, rewards have to be finite. However, it is clear that in the case of continuing tasks, the return of infinite trajectories will generally also be infinite, which would ultimately allow a comparison only via the sign of

the return. The solution to this is the application of a discount factor $\gamma \in [0;1]$ to the reward sequence, where the discounted return G_t is defined as follows.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.4}$$

Discounting models the priority that immediate rewards have over future rewards. with the reward R_{t+1} contributing $\frac{1}{\gamma^k}$ more to the overall return than the same reward obtained k timesteps later. This is yet again a natural concept, receiving 1€ today is (typically) preferable over receiving it next month. The edge cases $\gamma = 0$ and $\gamma = 1$ are special. The former yields a short-sighted agent that seeks out the highest immediate reward with every taken action and thus acts without any foresight, which is usually not desirable. The latter is equivalent to the undiscounted return 2.3. In all other cases $0 < \gamma < 1$, the discounted return of an infinite reward sequence is a sum of an infinite geometric series, and therefore indeed finite, with the rewards bounded by $\pm r_{\max}$.

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \leq \sum_{k=0}^{\infty} \gamma^k r_{\max} = \frac{r_{\max}}{1-\gamma} \tag{2.5}$$

While averaging the rewards of infinite trajectories is also a viable way of combating infinite returns, discounting is the common approach.

The two presented definitions of the return G_t, using additive and discounted rewards respectively, are the only sensible maximization targets, given the natural assumption of stationary preference. If an agent prefers the state sequence $\{S_0, S_1, S_2, \dots\}$ over $\{S'_0, S'_1, S'_2, \dots\}$, then $S_0 = S'_0$ should not affect the agent's preference of $\{S_1, S_2, \dots\}$ over $\{S'_1, S'_2, \dots\}$. Meaning, if an agent's preference is said to be stationary, the agent will always decide for one future over another, regardless of when it is allowed to make this decision. In order to unify both definitions of G_t and use a common notation for both episodic and continuing tasks, we can assume that episodes terminate in an absorbing state, which continues to transition to itself indefinitely without yielding any additional rewards, i.e. $\sum_{t=T}^{\infty} R_{t+1} = 0$. Therefore, the unified G_t is defined as follows, with $T = \infty$ and $\gamma = 1$ as mutually exclusive possibilities.

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{2.6}$$

Finally, the relationship between returns at successive timesteps is an essential aspect of RL. By unrolling 2.6, we can observe the following recursion.

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.7}$$

Since the agent's goal is to maximize G_t in expectation, it is tightly coupled to the value function, which in turn relates to the agent's policy, both discussed at the end of the

previous section. Within the MDP framework, the value function is formalized as the expected return G_t, either when the agent starts in state s, or takes action a in state s. This separates state-value functions $v_\pi(a)$ from action-value functions $q_\pi(s,a)$, both for the given policy π. Formally, a policy is a function that defines for every state $s \in \mathcal{S}$ a probability distribution over actions $a \in \mathcal{A}(s)$, i.e. $\pi(a \mid s)$ denotes the probability that the agent chooses $A_t = a$, if $S_t = s$. This formalization facilitates deterministic and stochastic policies. The two aforementioned value functions are thus defined as follows:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \,\middle|\, S_t = s\right] \tag{2.8}$$

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \,\middle|\, S_t = s, A_t = a\right] \tag{2.9}$$

Note that the expectations rely on the agent following a given policy π, which is indicated by the subscript. The recursive relationship of the return can be used to establish similar recursions for both value functions, which are called Bellman equations, named after Richard Bellman [8].

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} \mid S_t = s] + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} \mid S_t = s] + \gamma \mathbb{E}_\pi[v_\pi(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma v_\pi(s')], \quad \forall s \in \mathcal{S} \end{aligned} \tag{2.10}$$

In the second line we apply Equation 2.7, in the third line the properties of expectations $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$ and $\mathbb{E}[kX] = k\mathbb{E}[X]$. For action-values, the derivation is analogous, but the current action a is already known and a sum over all possible next actions a' has to be added.

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \\ &= \dots \\ &= \mathbb{E}_\pi[R_{t+1} \mid S_t = s, A_t = a] + \gamma \mathbb{E}_\pi[q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma \sum_{a'} \pi(a' \mid s')\, q_\pi(s', a')], \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s) \end{aligned} \tag{2.11}$$

These two Bellman equations express the relationship between current state- or action-values and the values of successive states or actions. In words, Equation 2.10 says that the state-value is defined as the sum of the expected immediate reward and the (possibly discounted) expected value of the next state. Equation 2.11 is analogous for action-values. These so called backup operations that send information about values from successor states/actions back to its preceding state/action are a core element of RL, and backup diagrams visualize these operations. The root node of a backup diagram represents the

value (of state or state-action pair) to be updated. The child and leaf nodes represent all transitions that affect that update via a reward or another estimated value. The top row of Figure 2.3 shows the backup diagram for v_π and q_π respectively. The Bellman equations are the foundation for various methods of learning v_π or q_π, because the value functions are the unique solutions to their respective Bellman equations. We can of course relate v_π and q_π to each other as follows:

$$\begin{aligned} v_\pi(s) &= \sum_a \pi(a \mid s)\, q_\pi(s,a), \quad \forall s \in \mathcal{S} \\ q_\pi(s,a) &= \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma v_\pi(s')], \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s) \end{aligned} \tag{2.12}$$

The value of a state is the expected action value over all possible actions from that state. The second equation then follows directly from the first one, by substituting into Equation 2.11.

As already said, both value functions are specific to the agent's policy. An optimal policy π_*, which the agent aims to find, has value functions v_* and q_* that return greater or equal values for all states and actions, than value functions for all other policies π'. While there can be multiple optimal policies, they all share the same value functions, which are referred to as the optimal state- and action-value functions. They are defined as follows:

$$v_*(s) \doteq \max_\pi v_\pi(s), \qquad \forall s \in \mathcal{S} \tag{2.13}$$

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a), \qquad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s) \tag{2.14}$$

Of course, v_* and q_* also have to satisfy the recursive self-consistency conditions given in Equations 2.10 and 2.11, but due to their optimality, these conditions can be rewritten without referencing a given policy π. In this case, they are called Bellman optimality equations.

$$v_*(s) \doteq \max_a \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma v_*(s')], \qquad \forall s \in \mathcal{S} \tag{2.15}$$

$$q_*(s,a) \doteq \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma \max_{a'} q_*(s', a')], \qquad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s) \tag{2.16}$$

Instead of using the expected value for a given policy, the equations simply maximize over the best action from s or s', which can be visualized in the backup diagrams as well, see bottom row of Figure 2.3. If the optimal value function v_* is found, an optimal policy π_* chooses in each state an action that will yield the highest possible value, without any additional look-ahead, since future consequences are already accounted for in the value function itself. Finding q_* has the added benefit of not needing any knowledge about the dynamics of the environment, because all state-action pairs already have assigned values and possible successor states need not be considered.

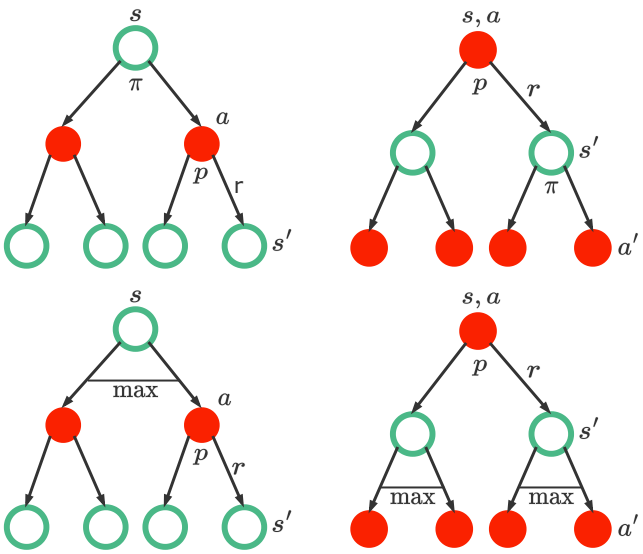


Figure 2.3: Backup diagrams for v_π, q_π (top row, from left to right), v_* and q_* (bottom row, from left to right).

In theory, finding a solution to the Bellman optimality equations also solves the RL problem itself. However, such an approach would rely on various assumptions that are usually not all fulfilled, such as the Markov property, knowing the exact environment dynamics, or having enough computational resources. Furthermore, the theories discussed in this section assume finite MDPs, albeit they are still applicable to general MDPs. In the case of small state sets, approximating value functions can be done using tables with entries for each state, hence this case is called the tabular case. However, state spaces are commonly too large for tabular methods to be applicable, giving rise to a need for parameterized function approximations. So while the theories of optimality discussed in this section are an important foundation for RL, they present an ideal that the agent seeks to approximate as closely as possible. The field of RL is generally concerned with problems, where finding optimal solutions is infeasible.

Notice that the MDP framework is generic and flexible, and is therefore applicable to a wide variety of RL problems. It suggests that any process of goal-directed learning can be reduced to the interaction of three signals, i.e. actions, states and rewards, visualized in Figure 2.2. The timesteps dictating this interaction do not have to be periodic or connected to time at all, they simply represent successive stages. Actions and states are abstract concepts. Their design depends on the specific RL problem and is thus a separate issue that is unrelated to the actual decision-making. To an agent that learns to play chess, an action can be the specific motion of a robotic arm, or the high-level decision of which move to make. A state can be raw camera input, or a representation of the chess board as a mapping from tile to piece.

In the next section, we will discuss a set of concrete algorithms for computing optimal

policies, called dynamic programming. These methods assume complete knowledge of the environment model and necessitate vast computational resources in practice, making them infeasible for a practical implementation. Nonetheless, comprehension of these methods is paramount, in order to understand the approaches we discuss and apply in the practical part of this work.

2.4 Dynamic Programming

Given a model of the environment as an MDP, with its dynamics represented by a complete four-argument function p (Equation 2.1), an optimal policy can be computed using methods from the set of Dynamic Programming (DP) algorithms. As mentioned in the penultimate paragraph of the previous section, these algorithms have little usefulness. However, it can be argued that all practical RL methods strive to emulate DP, without its assumptions and with a smaller computational footprint. Again, the subsequent concepts assume finite MDPs, but the ideas can be applied even in the case of continuous state and/or action spaces.

In the previous section, we concluded that an optimal policy π_* can be simply constructed, if an optimal state- or action-value function, v_* or q_*, was found. These value functions satisfy the Bellman optimality Equations 2.15 and 2.16 respectively. At the core of DP is the idea of iteratively updating an estimate of a value function, by turning these equations into assignment statements. Iteratively improving the value function then in turn leads to an iteratively improving policy, until reaching optimality.

The first key concept of DP is policy iteration, which consists of alternating steps of policy evaluation and policy improvement.

Policy evaluation refers to computing a value function, given a policy. The Bellman Equation 2.10 presented in the previous section shows that with a complete four-argument function p and a known policy π, $v_\pi(s)$ is the solution to a system of n linear equations in n unknowns, where $n = |\mathcal{S}|$. With large state spaces, iterative approaches are a more sensible choice than direct solution methods. Starting from a random initial value function approximation v_0 (except for terminal states, which have value 0), the Bellman equation is used as an assignment statement to update each successive approximation v_1, v_2, …, v_k, as follows:

$$v_{k+1}(s) \quad = \quad \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_k(s') \right], \quad \forall s \in \mathcal{S} \tag{2.17}$$

For each state s, the assignment sets the new value of s to be the expected sum of the immediate reward and the discounted old value of the succeeding state, along all single-step transitions that can occur using the evaluated policy. The Bellman equation declares $v_k = v_\pi$ to be a fixed point of this assignment. Policy evaluation continually sweeps over all states in the state space and applies it to each states' value to produce an updated approximation of the value function for the given policy. In general, this iterative policy evaluation is known to converge to v_π in the limit. In practice, the

algorithm should stop once the difference between new and old values drops below a certain threshold. The analogous expected update for action-value approximation is:

$$q_{k+1}(s,a) \quad = \quad \sum_{s',r} p(s',r \mid s,a)\,[r+\gamma \sum_{a'} \pi(a' \mid s')\, q_{k+1}(s',a')], \quad \forall s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{2.18}$$

Policy improvement refers to creating a new, improved policy π', using the value function v_π (or q_π) of a given policy π. This can be easily done, by always picking a greedy action in each state. An action is called greedy, if its value is at least as high as the value of all other possible actions from a specific state. Clearly, there can be multiple greedy actions with equal values in each state. A policy that resorts to taking greedy actions is said to act greedily, and is referred to as a greedy policy, with respect to the value function. Formally, $\pi'(s)$ is constructed as follows, using Equation 2.12:

$$\begin{aligned} \pi'(s) \quad &\doteq \quad \arg\max_a q_\pi(s,a) \\ &= \quad \arg\max_a \sum_{s',r} p(s',r \mid s,a)\,[r+\gamma v_\pi(s')], \quad \forall s \in \mathcal{S} \end{aligned} \tag{2.19}$$

That π' is at least as good as the given policy π is guaranteed via the policy improvement theorem. It stipulates that if for two deterministic policies π and π' the inequality $q_\pi(s,\pi'(s)) \geq v_\pi(s)$ holds for all states $s \in \mathcal{S}$, then π' must be at least as good as π, i.e. $v_{\pi'}(s) \geq v_\pi(s)$ must hold for all states $s \in \mathcal{S}$. If for a state s the former inequality is strict, the latter inequality is also strict. Of course, the definition of a greedy policy (Equation 2.19) satisfies the condition of the policy improvement theorem. Furthermore, policy improvement will always yield a strictly better policy, unless the given policy is already optimal. This becomes clear when replacing arg max with max in Equation 2.19:

$$v_{\pi'}(s) \quad = \quad \max_a \sum_{s',r} p(s',r \mid s,a)\,[r+\gamma v_\pi(s')], \quad \forall s \in \mathcal{S} \tag{2.20}$$

Assuming that the new policy π' remained as good as π, as opposed to being strictly better, we can set $v_{\pi'} = v_\pi$. However, this turns Equation 2.20 into the Bellman optimality Equation 2.15, i.e. π' and π would both be optimal policies, and $v_{\pi'} = v_\pi = v_*$.

As already mentioned, policy evaluation and improvement can be naturally combined into policy iteration, yielding a sequence of policies and value functions that improve monotonically. Each new policy is either already optimal, or strictly better than its predecessor. Since we assume the MDP is finite, the number of possible policies is also finite, thus guaranteeing convergence to an optimal policy in a finite amount of steps.

As opposed to starting each policy evaluation with a random initial v_0, the value function of the previous policy can be reused, thus greatly increasing the speed of convergence. Nevertheless, the policy evaluation might still require many sweeps through the states of the environment to reach its goal of a barely changing value function. However, there are various ways of shortening this step of policy iteration, while still maintaining its guaranteed convergence. One method is the so called value iteration algorithm, which

truncates policy evaluation after finishing a single update to each state, i.e. one sweep. It uses the following assignment as an update rule (again, analogous for action-values):

$$\begin{aligned} v_{k+1}(s) &= \max_a \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma v_k(s')], \quad \forall s \in \mathcal{S} \\ q_{k+1}(s, a) &= \sum_{s',r} p(s', r \mid s, a)\,[r + \max_{a'} \gamma q_k(s', a')], \quad \forall s \in \mathcal{S}, a \in \mathcal{A}(s) \end{aligned} \tag{2.21}$$

These are the Bellman optimality Equations 2.15 and 2.16 in the form of an assignment. Value iteration iteratively applies this update to all states, with each iteration being essentially equivalent to a single sweep of policy evaluation and improvement respectively. Analogous to policy evaluation, value iteration arrives at v_* (or q_*) in the limit, but should stop when the changes to the value function are sufficiently small between sweeps. Finally, a good (or optimal) policy will act greedily with respect to the returned value function.

While policy iteration runs policy evaluation completely, and value iteration runs it just once, there is a whole set of truncated policy iteration algorithms that lie between these extremes, utilizing varying amounts of evaluation sweeps. An even more granular approach offer asynchronous dynamic programming algorithms. As opposed to running sweeps over the entire state set, which can quickly become infeasible in practical settings, these algorithms are based on the idea of updating values asynchronously, i.e. in no particular order. Additionally, while states cannot be ignored fully, some may be updated more often than others, e.g. due to their importance. This also allows running asynchronous DP algorithms simultaneously with the agent's interaction with the MDP environment, establishing a bidirectional influence. The algorithm applies updates to the states visited by the agent, and the agent uses the current value function and policy when making decisions. Effectively, states that are more pertinent to the agent will be focused on, and will thus have more accurate value estimates, which is a common approach in RL methods. To ensure convergence, all states must be updated eventually.

Independently of their differences, these variations are summarized by the term Generalized Policy Iteration (GPI), describing the general concept of interacting processes of policy evaluation (adapting the value function with respect to the current policy) and policy improvement (making the policy greedy with respect to the current value function). In a way, these two processes cooperate and compete simultaneously, both aiming towards different goals. Policy evaluation typically causes the current policy to not be greedy anymore, and policy improvement causes the current value function to be incorrect. However, each step leads closer to the overarching goal of optimality. A stabilization of both processes means that the found policy is greedy towards its own value function determined via evaluation, implying optimality. The general concept of GPI, as well as the push-pull relationship we just described, is visualized in 2.4. Usually, RL methods can be described as some form of GPI. Lastly, a key aspect of the DP methods is their use of bootstrapping, which is the concept of using the estimated value of the next state to update the estimated value of the current state. Bootstrapping is prevalent in many RL methods.

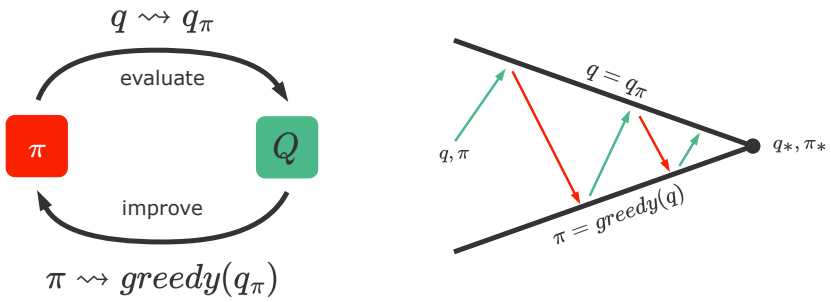


Figure 2.4: GPI constitutes alternating steps of policy evaluation and improvement (left). These two processes cooperate and compete at the same time (right).

As already mentioned, the requirement of having full knowledge of the environment dynamics (via Equation 2.1) is a crutch. In the next section, we will cover algorithms that learn from past experiences alone, in the form of complete or partial trajectories, without additional information about the MDP underpinning the environment. As opposed to DP, value functions and policies are learned, instead of computed. This approach, and thus also the following algorithms, is at the core of RL. Furthermore, notice that the lack of a model prohibits using Equation 2.12 to obtain action values from state values. Therefore, directly focusing on learning q_π is much more useful, which also greatly simplifies the construction of a greedy policy, see 2.19. We will thus primarily discuss learning action-value functions from now on.

2.5 Learning Methods

2.5.1 Monte Carlo Methods

Assuming tasks are episodic, one approach to reinforcement learning is utilizing sample returns of experienced (or simulated) episodes. Methods that follow this approach are called Monte Carlo (MC) methods. We established earlier, in Equation 2.8, that the value of a state-action pair (s, a) is the expected return obtained from taking action a in s, and following π thereafter. MC methods estimate this expectation, by averaging over the experienced returns after visiting the node (s, a) in a trajectory. With increasing amounts of experience, this estimate converges to the true expectation. Since returns become available on termination, MC methods perform updates to the estimated value function and policy on an episode-by-episode basis. All steps of the trajectory following the node (s, a) affect the return and are thus updated. Just like for DP, we visualize this in a backup diagram, see Figure 2.3. This is in contrast to the DP algorithms presented before, where updates are applied on a step-by-step basis. Nonetheless, MC methods still follow the GPI loop presented in the previous section. Clearly, MC methods do not bootstrap, because the estimates of action-values do not depend on existing estimates of

other actions or states.

One distinct advantage of the MC approach is the fact that more computational resources can be expended on learning values for states and actions that are of particular interest, by focusing on generating trajectories that start at that point. This is entirely independent of the dimensions of the state-/action- space. However, while DP algorithms operate with sweeps of the entire space, MC methods must ensure that all state-action pairs are experienced at some point. Always concentrating on the currently best values might lead the agent to disregard potentially better states/actions, due to imprecise or plainly wrong value estimates, caused by a lack of experience involving those states/actions. This is an important topic in RL in general, known as the exploration-exploitation trade-off. Essentially, it stipulates that an agent must strike a balance between behaving in a manner that appears to be best, and also experiencing nominally worse states/actions, which might ultimately lead to higher returns. There are two possibilities of forcing the agent to perpetually explore, dividing RL methods into two categories. Namely, on-policy and off-policy methods. The former improve on the same policy that is used during interactions with the environment. That policy must have a built-in exploration mechanism that occasionally forces it to take sub-optimal decisions. Typically, so-called ϵ-greedy policies are used, which usually act optimally, but will act randomly with a small probability ϵ. This approach still falls into the category of GPI, since the requirement is moving towards a greedy policy, but not necessarily fully reaching it. On the other hand, off-policy methods rely on a behavior policy for decision making, while using the obtained experience to learn a deterministic, greedy target policy. In contrast, the behavior policy can engage in exploration. A key assumption is that any action that occurs by following the target policy must have a non-zero probability of being taken by the behavior policy, for instance by using an ϵ-greedy policy. The off-policy approach is the more general concept, encapsulating the on-policy approach with the case of behavior and target policies that are equivalent. Furthermore, off-policy methods offer a way to learn from experiences supplied by a human expert or a non-RL system. It is important to note that the episode returns obtained by using the behavior policy must be weighted with the ratio of the probabilities of the actions taken during the episode to be taken under the behavior and target policies. This ensures that the sampled returns actually approach the expectation for the target policy, as opposed to the behavior policy. This is called importance sampling, and there are various approaches.

While we ignore the details of MC algorithms, we examine a key part, namely the incremental update of action values. Instead of storing all experienced returns for a given state-action pair (s, a), and computing the value $q(s, a)$ as the average (possibly weighted,

due to importance sampling), we can employ incremental updates as follows:

$$
\begin{aligned}
Q_{t+1} &= \frac{1}{\sum_{k=1}^{t} w_k} \sum_{k=1}^{t} w_k G_k \\
&= \frac{1}{\sum_{k=1}^{t} w_k} \left(w_t G_t + \Big(\sum_{k=1}^{t-1} w_k\Big) \frac{1}{\sum_{k=1}^{t-1} w_k} \sum_{k=1}^{t-1} w_k G_k\right) \\
&= \frac{1}{\sum_{k=1}^{t} w_k} \left(w_t G_t + \Big(\sum_{k=1}^{t-1} w_k\Big) Q_t\right) \\
&= \frac{1}{\sum_{k=1}^{t} w_k} \left(w_t G_t + \Big(\sum_{k=1}^{t} w_k\Big) Q_t - w_t Q_t\right) \\
Q_{t+1} &= Q_t + \frac{w_t}{\sum_{k=1}^{t} w_k} (G_t - Q_t) \qquad t \geq 1
\end{aligned}
$$

The weights w_k are either just 1 for all k, or depend on the kind of importance sampling applied. This incremental update allows us to update estimates after each return, without storing all returns experienced before. Assignments of the following form are very common in RL:

update.py

```
new_estimate = old_estimate + weight * (target - old_estimate)
```

Algorithm 2.1: Typical RL update assignment.

2.5.2 Temporal-Difference Learning

Next, we will discuss Temporal-Difference Learning (TD), a concept that the authors of [83] hail as cardinal. Again, TD follows the principle of GPI. TD relies both on sampling past experiences to learn from, as in MC methods, and on employing bootstrapping, as in DP. It can thus be viewed as a conceptual blend between the two approaches.

TD utilizes a recursive relationship we have established in Equation 2.11. To repeat:

$$
\begin{aligned}
q_\pi(s,a) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] && (1) \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] && (2)
\end{aligned}
$$

MC methods gather sample returns to estimate the expected return (1) and use it as a target. DP uses an estimate of the recursive relationship (2) as a target. As opposed to MC, DP can utilize its full environment model to compute the expectation, but the current estimate $Q(S_{t+1}, A_{t+1})$ is used in place of the unknown $q_\pi(S_{t+1}, A_{t+1})$. Similarly, TD

also uses (2) to estimate the target, by both sampling the expectations and bootstrapping with an estimated Q. While MC methods have to wait until the end of an episode to form a target with the obtained reward, TD methods can form a target and run updates at every timestep. The update assignment follows the form of 2.1:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big] \tag{2.22}$$

The difference between the target and the current estimate in square brackets, i.e. the so-called TD-error, occurs in different forms across the entire RL domain. In general, TD has significant advantages over MC. Long episodes prolong the waiting time of MC methods, and continuous tasks render them useless entirely, whereas TD methods work incrementally. Exploratory actions have a much stronger effect on the speed of MC learning, while TD uses each transition without taking subsequent actions into regard. Nevertheless, without going into detail, convergence to q_π is still guaranteed for TD under certain assumptions. And albeit not proven formally, learning usually converges faster with TD than with MC.

Again, we can differentiate between on-policy and off-policy approaches to TD. An on-policy method built around the GPI framework is called "Sarsa", named after the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ that makes up a transition and is used in the TD update. Sarsa constructs this quintuple in its entirety, before updating Q. It uses a policy π_Q derived from its current action-value estimate Q to select an action a, based on the current state s. It then immediately takes the action and observes the environment's response, in the form of the reward r and the next state s'. Finally, it uses the same π_Q to pick the next action a', based on the received information. Only then, it updates Q and consequently also π_Q. As discussed in the previous section, convergence is guaranteed, as long as it is ensured that the policy visits all state-action pairs infinitely often in the limit, which can be achieved with an ϵ-greedy policy, for instance. On the other hand, the off-policy counterpart to Sarsa is called Q-learning. Instead of selecting a next action A_{t+1} and using its value as part of the target, Q-learning chooses the action a that yields the highest value. The TD update assignment thus becomes:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\big] \tag{2.23}$$

Since this breaks the dependence on the current behavior policy, Q-learning is deemed an off-policy algorithm. Again, we visualize the TD updates of both Sarsa and Q-learning with backup diagrams in Figure 2.5.

Due to the max-operation used in Q-learning and implicitly also in Sarsa, via the ϵ-greedy policy, the estimated values in Q can have a significant upward bias, called maximization bias. For instance, the true action values for all actions in a state s could be 0, as is the case for terminal states. The estimated values $Q(s, a)$ for all $a \in \mathcal{A}(s)$ will be uncertain and scattered around 0, but the maximum over these estimates will be positive, thus leading to the aforementioned bias. This bias can be broken with a technique called double learning. As the name suggests, double learning uses two symmetrical estimates,

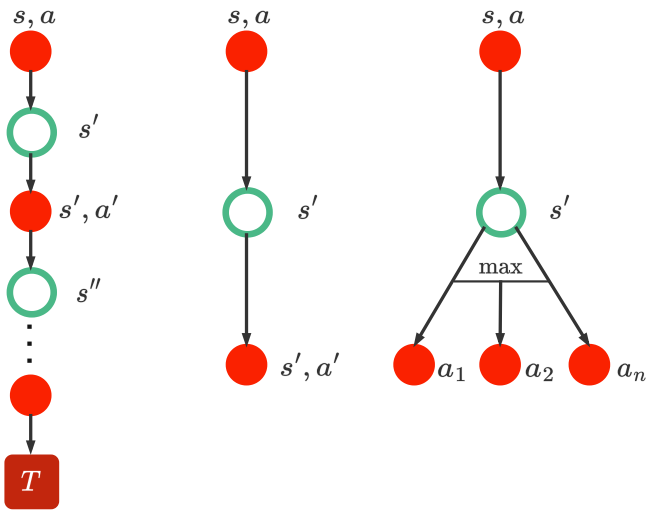


Figure 2.5: Backup diagrams for MC methods, Sarsa and Q-learning (from left to right).

Q_1 and Q_2. While one is used to obtain the action with the highest value, the actual value of that action is taken from the other estimate. In other words, the max-operation is separated into action selection and action evaluation. For each update step, one of them is randomly chosen to be adapted. For example, if Q_1 is selected, the update assignment looks as follows:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q_1(S_{t+1}, \arg\max_a Q_2(S_{t+1}, a)) - Q_1(S_t, A_t)\right] \quad (2.24)$$

Double learning not only converges faster by countering the maximization bias, it can also lead to better performance at asymptote.

2.5.3 n-Step Temporal-Difference Learning

n-step TD methods bridge the gap between the one-step TD and the MC methods discussed in the previous subsections. They are a seamless unification of the two approaches, offering a way to shift arbitrarily close to one or the other. To repeat, while MC uses the entire episode return as an update target, TD uses the one-step return, i.e. the sum of the currently received reward and the discounted expected future return. This (extreme) case of TD is also called TD(0). Of course, the action-value estimates are used in-place of the future return. However, as the name suggests, it is also possible to use TD with the two-step, three-step, or n-step return as an update target. Over n timesteps, actions are taken and rewards received. Ultimately, the update target is constructed as the sum of all n received rewards, and the appropriately discounted expected future return. Concretely, the one-step return used in TD(0) and the n-step return are defined

as follows:

$$\begin{aligned} G_{t:t+1} &\doteq R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}) \\ G_{t:t+n} &\doteq R_{t+1} + R_{t+2} + ... + R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) \end{aligned} \tag{2.25}$$

Instead of updating at every timestep, n-step TD permits updates after an arbitrary amount of steps n. The corresponding backup diagram would therefore lie somewhere between those of MC and TD, as seen in Figure 2.5.

2.6 Function Approximation

So far, we have discussed a spectrum of learning methods, spanning from MC methods to TD. These methods differ from DP, due to their use of experience to "learn" a value function, as opposed to known environment dynamics to "compute" it. All of these methods use backup operations to estimate action value functions, and follow the pattern of GPI, i.e. the idea of iteratively improving estimates of a value function and a policy, based on each other. However, we have also identified key components of variation. For example, on-policy and off-policy methods, synchronous and asynchronous updates, or the action selection process and the accompanying exploration-exploitation trade-off. Nevertheless, as briefly mentioned in Section 2.3, tabular methods rapidly lose utility, once the size of the state-space (and/or action-space) grows to a point, where a practical implementation becomes infeasible, both due to the speed of convergence, as well as the required computational resources and memory. Unfortunately, this point is reached by all but the most primitive RL problems, such as an agent navigating a 2-dimensional maze with 10,000 states. For instance, even a comparatively simple game such as chess encompasses around 10^{40} states. While it is immediately clear that the methods discussed in this chapter cannot be applied in that case, it is also not intuitive. The requirement of having to experience a specific state multiple times before being able to select a good action in that state has no basis in reality. Generally speaking, after learning the rules of the game, humans play chess by generalizing from previously experienced situations to those they have not seen yet. Since ambitions of reaching optimality are futile in RL problems of this dimension, the aim shifts to finding good approximate solutions. To that end, we must use function approximation, which essentially is the idea of replacing lookup tables with other representation methods. The goal of function approximation is to compress the state-space representation to a much smaller size, which in turn allows the agent to inductively generalize from visited to unknown states. The name stems from the fact that examples of the target function, in our case an action-value function or a policy, are used to learn an approximation of the true function. Clearly, this incorporates an instance of SL into RL, as we have mentioned in Section 2.2. Function approximation is an umbrella term for a wide variety of approaches, including state aggregation and linear methods. However, we will focus on non-linear methods involving LDNNs. This combination of DL and RL, i.e. DRL, was responsible for the range of state-of-the-art RL methods that are being used in practice. We will present some of these methods in Chapter 4 and apply them in the practical part of this thesis, presented in Chapter 7.

CHAPTER 3

Fundamentals II: Recommender Systems

This chapter covers the fundamentals of Recommender System (RS). It will provide a basic understanding of RSs, an overview of various recommendation and evaluation methods, as well as the key challenges of sparsity and cold-starts. The sources for this chapter are primarily [71] and [3], along with the corresponding entries for RSs in [74] and [70].

3.1 Introduction to Recommender Systems

Due to the rapid rise of the world wide web, there has been an exponential increase in the amount and variety of information available online. Along with globalization and a boom of digital commerce and services, this has led to an overwhelming amount of possible choices users can make. From the thousands of movies and shows to watch on Netflix [81], to the millions of songs and podcasts to listen to on Spotify [80], up to the huge variety of products and product categories to buy at Amazon. Instead of being beneficial, this sheer overload of information was shown to have negative effects on the well-being of users [77]. RSs have emerged as a software solution to this problem, stemming from the observation that people often rely on recommendations to arrive at a decision.

In accordance with the understanding of the term AI that we have established in Section 2.1.1, we can regard RSs as rational agents. The goal of an RS is to infer the interests of users, and use that information to serve them recommendations for items that they would want to interact with. The specifics of these systems are highly dependent on the domain and the available data. Furthermore, there is a multitude of recommendation methods, using techniques from several intersecting fields, such as statistics, data mining, information retrieval and ML. The inherent commercial utility and the user-centered

nature of RSs entails that they play a very large role in people's everyday lives. Amazon recommends products [33], Spotify recommends music [28], Netflix recommends movies and shows [82][13], Instagram recommends photos and videos [29], and so on. While the data used in these examples spans a wide variety of domains, requiring vastly different approaches to analyzing and processing the data, they all fit into the overarching RS framework. According to the authors of [71], an RS consists of three main elements, namely a collection of users and a collection of items, as well as the relations between two objects from each respective collection:

1. *Items*: An item is the object of interest that the RS recommends to users. For instance, following the examples above, an item could be a specific, product, movie, or song. Clearly, there is a broad spectrum of possibilities for encoding an item, ranging from a simple identification number, a combination of extracted features, to a latent representation learned by an LDNN, or a node in a knowledge graph about the entire domain. Any available data can be used for the encoding. For the example of a movie, one could use the genre, the title, the list of actors involved, the director, etc. Furthermore, each recommended item has an attached value that is dependent on the user receiving the suggestion. The sign of the value describes whether the user's reaction to the recommendation was positive or negative.

2. *Users*: In order to achieve its goal of generating tailored recommendations, an RS must utilize the available data to create a user model. Again, the details of this model depend on the domain and the accessible information. For instance, in the case of social media platforms, one could use sociodemographic attributes such as age, gender, location, relationship status, etc. to construct a user encoding. In other cases, when users have not willingly provided this information, a user's behavior within the domain can serve as a basis for a user model. For example, the order history, search history, browsing patterns, reviews, etc. of an Amazon user can be accumulated into an encoding. Last but not least, the relations between individual users in the collection can offer valuable information. An RS could suggest items to user u due to their connection to another user v, be it friendship, kinship or a relationship of trust, e.g. when user v is a celebrity, or an influencer.

3. *Transactions*: Ultimately, the key element of an RS are the binary interactions between individual users and specific items, called transactions. An RS must keep a log of this transactional data that occurs during human-computer interaction. Aside from a unique reference to the respective user and item model involved in the transaction, contextual data can be collected as well. Examples of such data are timestamps, or the query that led the user to interact with the item. However, the arguably most important information is user feedback. It can be explicit, e.g. in the form of a numerical (1-5 stars), ordinal ("[strongly] agree", "neutral", "[strongly] disagree"), or binary ("interesting", "uninteresting") rating. Unary ratings also exist, where users can express a positive sentiment towards an item, but not a negative one, such as via "likes" on Instagram. It is also possible, and often necessary, to

collect and rely on feedback that is implicit. Instead of having access to directly expressed user opinions, the RS must infer a user's interests and preferences, based on their actions. Which products a user clicked on or bought, which songs a user listened to, and which movies a user watched, are all examples of implicit feedback. Notice that all of these activities indicate interest, but not a positive sentiment. The user might not have liked the product, song, or movie. Furthermore, just like in the case of unary ratings, not taking a certain action is usually not an indicator for disinterest. For instance, a user might enjoy a movie or a song, but they are simply not familiar with it, which is a common occurrence, due to the collection of items typically being very large.

In essence, the goal of an RS is the ability to compute accurate predicted evaluations for a pair (u, i), where $u \in U$ denotes a user from the collection of users, and $i \in I$ denotes an item from the collection of items. In other words, an RS must implement a function $\hat{R} : U \times I \mapsto \mathbb{R}$ that estimates the real underlying function R. Again, the details of this process are domain- and data-dependent. Furthermore, the predicted evaluation of an item for a user might depend on contextual variables as well, e.g. the user's location, the current time, etc. Generally speaking, the operation of an RS can ultimately be separated into three distinct tasks. Firstly, the RS must elicit user preferences. It does this by collecting and evaluating transactions, which describe the experience of a user with an item. Formally, such an experience is a tuple $(u, i, m, R(u, i, m))$, where m denotes the aforementioned contextual variables and $R(u, i, m)$ the user's (explicit or implicit) feedback. In addition, as already mentioned, the sociodemographic attributes of users can play a role in preference elicitation. Secondly, the RS uses the collected transactions to predict evaluations for new user-item pairs. There is a wide variety of recommendation methods that can be employed in this step. They can be categorized into different types, which we will discuss in the next section. Thirdly, the RS uses the predicted evaluations to finally generate recommendations. There are several approaches. For example, the RS can construct a list of the top k items sorted from highest to lowest predicted evaluation. It can also just predict a single item it deems best. Or, it can opt to rank all evaluated items, which usually is a set of candidate items, i.e. a subset of all items that were eligible for recommendation for one reason or another. However, note that while the predicted evaluation plays a central role in this process, there are other important factors as well, such as diversity and novelty. It is possible that the set of items with high predicted evaluations are all very similar, i.e. the recommendations are not diverse. It is also possible that the recommended items are very similar to items that the user has already interacted with, i.e. the recommendations are not novel.

Last but not least, it should be mentioned that not all RSs fall into the framework presented in this section. For example, a non-personalized RS is not concerned with specific users or preference elicitation at all, and simply aims to recommend items that will appeal to the average user. Another example are RSs that do not collect transactions, and instead recommend items based on specific queries provided by a user. Nevertheless,

the majority of RSs, as well as the RS discussed in the practical portion of this work, can be described as we did in this section.

3.2 Recommendation Methods

There is a broad spectrum of recommendation methods that can be employed to estimate predicted evaluations. These methods can be grouped into categories, mainly according to the data they utilize. In this section, we will briefly present the most important recommendation approaches. The following taxonomy was first introduced in [11].

- *Content-based Methods*: These methods focus on the attributes and features of items. Preference elicitation for a user is usually done on the basis of the encoded items that the user interacted with in the past, but other approaches are also possible, such as via the user's answers to predefined questions. Subsequently, the RS aims to infer the content of items that could be interesting to the user. Again, there are multiple possible approaches. Given encoded items, the RS could compute predicted evaluations, based on the learned user preferences. On the other hand, it would also be possible to directly generate an encoding for a pseudo-item, and recommend items with similar encodings. As an example, a content-based RS could recognize that a user is particularly fond of science-fiction and action movies, and thus recommend other movies from those genres.

- *Collaborative Filtering*: These methods revolve around the idea of inferring from previously occurred transactions. We can further distinguish memory-based and model-based methods.

  Memory-based Methods: These methods use neighborhoods to compute predicted evaluations. On the one hand, we can group together users with similar preferences. The similarity between two users is determined from each user's past transactions. Recommendations for one user are then based on the activities of other users in the neighborhood. In other words, the probability of user Alice enjoying a movie that received a highly positive rating by user Bob increases proportionally to how much overlap there is between their watch histories. On the other hand, we can group together items that have received similar ratings from the same users. Recommendations for a user are then based on the feedback they have given for other similar items. In other words, we can assume the movies A and B are similar, when both have received comparable ratings from a set of users, e.g. users Alice, Bob and Eve loved both movies, users Carol and Dan hated them. We can then infer that user Frank will enjoy movie A, because they have enjoyed movie B. Memory-based methods have the advantage of simplicity and explainability, but are lackluster when transactions are sparse. Meaning, the number of users and/or items is high, and/or the amount of feedback per user/item is low.

  Model-based Methods: These methods aim to build a statistical model for user feedback. While memory-based methods rely on a statistical similarity between

users/items, these models assume that the similarity is induced by latent factors in the data. A successful approach to obtaining a model is matrix factorization, whereby users and items are mapped to feature vectors along k latent dimensions, such that the inner product of a user-item pair matches any recorded feedback. This can be done e.g. via singular value decomposition. The predicted evaluation for a specific user-item pair can then simply be obtained by taking the inner product of the respective feature vectors.

- *Knowledge-based Methods*: These methods utilize, as the name suggests, concrete domain knowledge, stored in some form of knowledge base (e.g. a knowledge graph) to identify which properties of an item match which user needs. Naturally, these methods are particularly useful in domains, where transactions are rare, but more consequential, and items are complex. For example, real estate, cars, or financial services are bought comparatively rarely to products on Amazon, so there is not enough user feedback to effectively employ one of the other mentioned methods. Additionally, since buying a house or health insurance is a much more important decision than which movie to watch, users usually have specific constraints and conditions that must or should be met by the recommended items. We can further distinguish case-based and constraint-based methods:

  Case-based Systems: These systems require users to specify existing cases as a target. The attributes of the case are then used to identify similar items. The process can be interactive, whereby the user slightly modifies the recommended item and specifies that as the new target, in order to refine the recommendation. An example for a case-based system is a car RS, which allows users to define ideal values for certain attributes of the car (e.g. color, horsepower, price, etc.), and/or demand recommendations similar to a specific car.

  Constraint-based Systems: These systems allow users to set constraints, e.g. upper and lower values, specific values, value ranges, etc., which are then matched to items via domain-specific rules. Again, the process is meant to be interactive, with users tightening or relaxing their constraints, depending on the amount and quality of the recommended items.

- *Hybrid Methods*: Last but not least, hybrid methods attempt to utilize some combination of the preceding methods, with the goal of mitigating the disadvantages of a single method. For example, CF methods struggle to compute predicted evaluations for items that have occurred in few transactions. In contrast, content-based methods do not have this problem. An example for a hybrid approach is using both methods to produce lists of recommended items, and subsequently combining them using an adaptive weighted average, whereby items recommended by CF are weighted proportionally to the amount of available feedback for that item.

Analogous to the variety of possible ways to compute predicted evaluation and generate recommendations, there are multiple approaches to evaluating an RS. We will briefly cover the key aspects of RS evaluation in the next section.

3.3 Recommender System Evaluation

First and foremost, the evaluation of an RS can be conducted in the context of three experimental settings:

- *Offline Experiment*: These experiments use a historical dataset of collected transactions to train and evaluate an RS. Since transaction logs are typically time-series data, a common approach is to split the data into a training-, test-, and possibly development set, along certain points in time. A fundamental assumption is that the dataset is an accurate representation of its source upon deployment of the RS. Meaning, aspects such as user behavior or the set of active users should be similar in the real system. Naturally, offline experiments are cheap and easy to conduct, allowing to iteratively test, compare and finetune a multitude of methods, without having to expose real users to this process. The main focus of offline experiments is to measure the quality of predicted evaluations. However, the real effect of an RS and the response of the user base can only be measured accurately in the following two settings.

- *User Studies*: In order to collect more information about the performance of an RS, especially regarding metrics that are hard to measure, e.g. emotional user responses such as "satisfaction", one must move from offline experiments to more user-centric evaluations. User studies are controlled experiments, where a carefully recruited group of actual users is instructed to interact with the RS, typically via given tasks. During the interactions, quantitative and qualitative data can be collected. The former could consist of the number of completed tasks and the time taken for each one, the number of interactions with recommended items, or even eye tracking data. The latter could be obtained by questioning the subjects, about topics such as the user interface, their perceived interest in the recommendations, or how difficult the given tasks were. Obviously, user studies are difficult, time-consuming and expensive experiments, but can provide a very wide range of insights into the performance of an RS.

- *Online Experiments*: Finally, the most holistic evaluation approach is the deployment of an RS into an existing environment, where real users can interact with it. Usually, A/B testing is employed to measure the impact of an RS, whereby users are randomly separated into two groups. Subsequently, each group is exposed to a different system. One system could use a new RS, whereas the other system only serves random suggestions, or one could compare two different RSs. Due to the potential negative effects of a subpar RS on the user base, e.g. users might be discouraged to utilize the service in the future, online experiments are usually conducted at the final stage of RS development. Unfortunately, online experiments are often only accessible to the owners of the environment, typically a commercial system, e.g. Amazon, Netflix, or Spotify.

As already mentioned, offline studies focus on measuring the performance of an RS, for which there is a multitude of metrics available. The choice depends on the goals of the RS and the type of predicted evaluations. In the case of systems with explicit feedback, accuracy measures can be used to evaluate the errors between predicted evaluations and the actually collected rating, for transactions in the test set. Commonly used metrics are the well-known mean squared error, root mean squared error, or mean absolute error. In the case of implicit feedback, or unary ratings, the RS can be evaluated on the basis of which of the items recommended to a user did they actually interact with. For instance, whether a user listened to recommended songs, watched recommended movies, or clicked on recommended products. Of course, this information must be available in the dataset. Note that the data is generally collected while another RS, or none at all, was used. Unfortunately, this means we must assume that a user not interacting with an item implies that they were not interested, even if that item were recommended to them. Clearly, it is possible that the user was simply unaware of the item instead. The only way to relax this assumption, is by collecting information on which items the user has encountered, or was otherwise made aware of by the system. Evaluation based on occurred interactions can be done using metrics revolving around the number of true/false positives/negatives. Most notable are precision p, recall r, and false positive rate fpr.

$$p = \frac{tp}{tp + fp} \tag{3.1}$$

$$r = \frac{tp}{tp + fn} = tpr \tag{3.2}$$

$$fpr = \frac{fp}{n} = \frac{fp}{fp + tn} \tag{3.3}$$

These metrics present a trade-off, since recommending more items often increases recall and reduces precision. Therefore, it is usually beneficial to employ evaluation metrics that compare both values, such as a precision-recall curve, or receiver operating characteristic (ROC) curve. The former pits precision against recall, focusing on the proportion of recommended items that the user is interested in. The latter compares recall and the false positive rate, focusing on the proportion of uninteresting items that are nonetheless recommended to the user. Which curve to use depends on the domain and the inherent goals of the recommender system. A precision-recall curve is more suitable, when as many interesting items as possible should be recommended, attaching less relevance to the amount of uninteresting items in the set of suggested items. In contrast, a ROC curve is more suitable, when it is desirable that most of the recommended items are actually interacted with, attaching less relevance to the amount of interesting items that did not get suggested. In order to facilitate comparing several RSs, metrics that summarize the mentioned curves are useful, namely F1-measure and the Area Under the ROC Curve (AUC). While AUC is self-explanatory, the F1-score is defined as the harmonic mean between the equally weighted precision and recall, whereas the generic Fβ-score applies

a larger weight to the recall value, such that it is β times as important:

$$F_1 = 2 \cdot \frac{r \cdot p}{r + p} \tag{3.4}$$

$$F_\beta = (1 + \beta^2) \cdot \frac{r \cdot p}{r + (\beta^2 \cdot p)} \tag{3.5}$$

Because items are usually recommended to users in the form of lists, it is often useful to compute the evaluation metrics only for the top N suggested items, due to their prominence. In this case, or when the amount of recommendations is fixed to always be N, we instead evaluate e.g. precision at N, denoted precision@N. Note that these metrics disregard the order the recommended items are in. However, in practical applications, these rankings play an important role. It is natural for users to expect the recommendations to be ranked from most to least suitable for them. Therefore, an evaluation metric that measures how closely the ranking produced by the RS matches (one of) the ideal rankings provides useful insights. Again, there is a multitude of metrics available, emphasizing different aspects. A widely used measure is normalized discounted cumulative gain (NDCG), which applies a logarithmically increasing discount to each ranking position, commonly with a logarithm of base 2. Furthermore, NDCG is usually computed for the top N items, where $N = 5$ or $N = 10$ are typical values. The discounted cumulative gain for the top N suggestions is defined as follows:

$$\text{DCG@}N = \sum_{j=1}^{N} \frac{g_{u,i_j}}{\log(j+1)} \tag{3.6}$$

g_{u,i_j} is the "gain" the user u gets from item i, presented at position j in the ranking, i.e. a value denoting how relevant item i is to user u. This value can be a simple binary, as well, where $g_{u,i_j} = 1$ describes relevant, and $g_{u,i_j} = 0$ irrelevant items. Finally, NDCG normalizes DCG by the ideal DCG (IDCG):

$$\text{NDCG@}N = \frac{\text{DCG@}N}{\text{IDCG@}N} \tag{3.7}$$

The ideal DCG is the DCG for an ideal ranking, which contains relevant items sorted from most to least gain. Clearly, in the case of a binary gain value, there can be multiple ideal rankings. This occurs, for instance, when only user's implicit feedback is available, e.g. in the form of clicks. A closely related metric is the reciprocal rank (RR), which attaches a weight of $\frac{1}{k}$ to a recommended item that the user interacted with, where k denotes the item's position in the ranking, and sums over them. The mean reciprocal rank (MRR) is the average RR over all rankings constructed by the RS, i.e.:

$$\text{MRR} = \frac{1}{N} \sum_{j=1}^{N} \frac{1}{rank(i_j)} \tag{3.8}$$

While the predictive power of an RS is undoubtedly essential, there are other important factors beyond accuracy, which we have briefly mentioned in Section 3.1, such as novelty, serendipity, diversity, and trust. Again, the possibilities for evaluating an RS in regards to these properties are manifold. A discussion thereof is not in the scope of this thesis.

3.4 Challenges: Sparsity and Cold-Starts

In this section, we will briefly discuss two key challenges that usually occur when solving the recommendation problem, namely sparsity and cold-starts.

Sparsity describes the issue of scarcely received feedback, especially in the explicit form. In other words, users rarely provide feedback, the vast majority of users will never interact with the vast majority of items. The scarcity, or complete lack, of explicit feedback is an issue for all RSs. Collecting and utilizing implicit feedback is a way to mitigate this. Sparsity poses a major difficulty for RSs employing collaborative filtering techniques, due to the fact that it impedes the search for sets of users with comparable ratings. This effect is exacerbated by a high item-to-user ratio, i.e. a music RS will have to deal with significantly more sparsity than a movie RS, assuming the number of users is equal in both. Furthermore, sparsity will generally be very high in new systems that have yet to accumulate a good amount of feedback. Note that sparsity is less of an issue in content-based RSs, provided that sufficient content attributes are available.

Cold-Starts describes the introduction of new items or new users into a system, where the term "cold" denotes the shortage of initially available information. In the case of items, an absence of feedback for a specific item essentially prohibits a collaborative filtering method from recommending it. In the case of users, preference elicitation is hampered for users without any historical transaction data. Instead, an RS can rely on non-personalized recommendations during the cold-start period. Another possibility is to ask users to self-describe their preferences, for example by having them select desired genres of music, or their favorite artists or songs. Again, in content-based systems, cold-starts for items are not a problem, whereas cold-starts for users still have to be dealt with.

CHAPTER 4

Fundamentals III: Deep Reinforcement Learning

This chapter presents the set of DRL algorithms that we will apply in the practical portion of this thesis. These algorithms essentially cover all categories of DRL approaches, and include state-of-the-art methods. First, we will begin with a very brief history of DRL.

4.1 Brief History of Deep Reinforcement Learning

One of the earliest successful RL agents utilizing an ANN for function approximation was an agent playing backgammon, dubbed TD-Gammon [84]. Despite its limited knowledge of the game, the agent was able to reach a very high skill level, using a variant of TD. The ANN was trained via backpropagation of the TD-errors experienced by the agent. While this network was technically not "deep", one can argue that TD-Gammon marked the beginning of DRL [61]. Unfortunately, subsequent efforts to apply TD-gammon to other games were unsuccessful, fueling the belief that the approach was for one reason or another a fluke and would only work on backgammon [65]. The wave of research around deep learning that we briefly discussed in Section 2.1.3 had little effect on the field of RL, until Mnih et al. presented the Deep Q-Network (DQN) algorithm in papers published in 2013 [59] and 2015 [58]. The authors showed how a DQN agent learned to play classic Atari 2600 video games, by directly processing the raw pixels of game frames. The agent not only outperformed previous algorithms, but also reached the level of a skilled human across 49 games, without tuning the agent for each game. This breakthrough ignited the DRL research, leading to a multitude of novel DRL approaches in the years thereafter [23] [61] [95], some of which we will discuss in this chapter, starting with DQN.

4.2 DQN

The basic idea underpinning DQN is using a neural network, called the Q-network, as a function approximator for the optimal action value function q_*. Formally, this can be written as $Q(s, a; \theta) \approx q_*(s, a)$, where θ is a stand-in for the parameters and weights of the ANN. Prior work concluded that Q-networks are unstable and can diverge, due to three key issues [85]. Firstly, the correlations between subsequently experienced states. Naturally, a new state s' depends on the previous state s, and might be highly similar. Secondly, the correlations between the Q-networks outputs and the targets. And thirdly, the fact that even small changes to the Q-network can bring about significant changes of the corresponding greedy policy. In consequence, the assumption that the data used by optimization methods for learning is identically independently distributed cannot be upheld for experiences. Arguably the first successful approach to using ANNs for approximating the action-value function was NFQ (neural fitted Q-iteration) [72] [61]. By using large mini-batches of experiences, NFQ alleviated some of the aforementioned issues, without solving them. While this yielded good results in simple environments [61], the computational cost per iteration is proportional to the dataset size, making its use infeasible for more complex problems requiring larger Q-networks. Mnih et al. used two fundamental concepts to alleviate these problems, namely experience replay and a target network. Experience replay is the idea of storing experience tuples $e_t = (s_t, a_t, r_t, s_{t+1})$ in a dataset D_t, usually called the replay memory/buffer [61], at each time step t. Then, instead of using current experiences in the learning process, the agent samples mini-batches from D_t uniformly at random, i.e. $(s, a, r, s') \sim \mathcal{U}(D)$. This has the effect that correlations in the trajectories are broken up and changes in the data distribution are smoothed over. The target network is a second Q-network that, as its name suggests, is used in the computation of the update target. The target network lags behind the primary ("online") network, i.e. its parameters are frozen and only aligned to the online network's parameters every C steps. For instance, for the agent learning to play Atari games, the authors ran $C = 10000$ updates between each alignment. Otherwise, the two networks are architecturally equivalent. The use of target networks removes the correlation between outputs and targets, by using different network parameters for obtaining of each respective value.

Just like neural networks in other machine learning domains, Q-networks are trained by iteratively minimizing the losses between targets and outputs. Of course, there are many functions that can be used as the loss function $\mathcal{L}$, such as MAE-loss, MSE-loss, etc. For instance, using the latter, the loss function $\mathcal{L}_i(\theta_i)$ at iteration i is defined as:

$$\mathcal{L}_i(\theta_i) \quad = \quad \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(D)}\big[(y_i - Q(s, a; \theta_i))^2\big] \tag{4.1}$$

The expectation is taken over the uniformly sampled experience tuples. Unlike in SL, the targets y_i are not known beforehand. Instead, the target used in tabular Q-learning (Equation 2.23) is computed analogously, using the target network with the parameters θ_i^-, i.e.:

$$y_i \quad = \quad \mathbb{E}_{s'\sim\mathcal{U}(D)}\big[r + \gamma \max_{a'} Q(s', a'; \theta_i^-)\big] \tag{4.2}$$

Differentiation of the loss function with respect to the Q-network's parameters ultimately leads to the following gradient, which can be used for optimization via stochastic gradient descent:

$$\nabla_{\theta_i}\mathcal{L}_i(\theta_i) \quad = \quad \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(D)}\Big[(r+\gamma \max_{a'} Q(s',a';\,\theta_i^-)-Q(s,a;\,\theta_i))\nabla_{\theta_i}Q(s,a;\,\theta_i)\Big] \tag{4.3}$$

In the years after the release of the initial papers by Mnih et al., several improvements to DQN were introduced, which we will discuss in the following subsections.

4.2.1 Double DQN

We have already discussed the issue of overestimation in Q-learning in Section 2.5.2, as well as the solution to it, i.e. double learning. Incidentally, the same researchers that authored double Q-learning 5 years prior [36], presented Double DQN (DDQN) in [37], with the designated goal of embedding most of the advantages of double Q-learning into DQN, while changing the original algorithm as little as possible. Therefore, as opposed to following double Q-learning and introducing yet another Q-network, the target network is used. Concretely, the max-operation in the DQN target (Equation 4.2) is decoupled into action selection, done by the online network, and action evaluation, done by the target network. This makes sense, since the target network's parameters are held fixed for C steps at a time, thus providing more stable values. Unlike double Q-learning, which requires updating two estimates Q_1 and Q_2, the utilization of the target network means that the online network remains the sole network being trained. The original DQN algorithm is not adapted in that regard. In summary, Double DQN replaces the target Equation 4.2 with:

$$y_i \quad = \quad \mathbb{E}_{s'\sim\mathcal{U}(D)}[r+\gamma Q(s',\arg\max_{a'} Q(s',a';\theta_i);\,\theta_i^-))] \tag{4.4}$$

As mentioned in Section 2.5.2, double Q-learning not only mitigates the maximization bias, but can ultimately also lead to better performance overall. The authors of DDQN report the same effect for the agent playing Atari games, where their algorithm obtained significantly higher mean and median scores than the original DQN agent.

4.2.2 Dueling DQN

With the goal of designing a neural network architecture that is tailored for RL, the authors of [87] devised and presented the dueling network architecture. Again, the underlying DQN algorithm remains unchanged. Moreover, since Dueling DQN is only concerned with the Q-network's architecture, it can also be combined with DDQN, as well as other variants we will discuss later on. The dueling architecture utilizes a relationship between action-values $q_\pi(s,a)$ and state-values $v_\pi(s)$ that we have omitted in Chapter 2:

$$\begin{aligned} a_\pi(s,a) &\doteq q_\pi(s,a) - v_\pi(s) \\ a_\pi(s,a) + v_\pi(s) &\doteq q_\pi(s,a) \end{aligned} \tag{4.5}$$

$a_\pi(s,a)$ is called the advantage-function, since it represents how advantageous it is to choose action a, over selecting an action according to the policy π. Intuitively, by subtracting the value of a state from the value of taking a specific action in that state, we obtain a relative value of the action's importance. Furthermore, note that $\mathbb{E}_{a\sim\pi(s)}[q_\pi(s,a)] = v_\pi(s)$, which means that $\mathbb{E}_{a\sim\pi(s)}[a_\pi(s,a)] = 0$. The general concept of a separate advantage stems from [34], with its authors presenting a single advantage function later, in [35]. The dueling architecture explicitly splits the single stream of a conventional Q-network into two streams, estimating state values and action advantage values. Ultimately, these streams are combined in the network using Equation 4.5. Mathematically, this can be expressed as follows, with θ representing the parameters of the shared layers, while α and β denote the parameters of the advantage- and state-value layers respectively.

$$Q(s,a;\,\theta,\alpha,\beta) \quad = \quad V(s;\,\theta,\beta) + A(s,a;\,\theta,\alpha) \tag{4.6}$$

However, this equation is said to be unidentifiable, since the unique values of A and V cannot be obtained, if only Q is given. There are multiple ways to address this, with the authors opting for subtracting the mean of all advantage values from Q:

$$Q(s,a;\,\theta,\alpha,\beta) \quad = \quad V(s;\,\theta,\beta) + \Big(A(s,a;\,\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\,\theta,\alpha)\Big) \tag{4.7}$$

This leads to a more stable optimization process. While it also shifts V and A by a constant, whereby these estimates lose their initial meaning, the relative rank of the resulting Q remains unchanged.

Of course, the internal connections of a Q-network ensure that information is shared to some degree, i.e. learning about $Q(s,a_1)$ also entails learning about other action values $a' \in \mathcal{A}(s)$ in the state s. Nevertheless, especially in the case of a large number of actions having similar values, the use of the dueling architecture improves the agent's performance and sample efficiency. Again, the authors tested their architecture design using the familiar Atari benchmark, finding that it outperformed the original DQN on the majority of games.

4.2.3 Prioritized Experience Replay

The final key improvement to the original DQN covered in this section focuses on the sampling strategy for experience replay. Prioritized experience replay [76] aims to replay important experiences more frequently than others, as opposed to sampling uniformly at random from the replay memory. Concretely, a transition is characterized as more "important", the more the estimated value of an action differs from the target value. In other words, importance is directly proportional to the size of the TD-error (see Section 2.5.2). However, while it is good to prioritize experiences with large absolute TD-errors, it is not helpful to completely stop replaying "regular" experiences. To that end, the

authors propose the following sampling probability:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{4.8}$$

$$p_i = |\delta_i| + \epsilon \tag{4.9}$$

$$p_i = \frac{1}{\text{rank}(i)} \tag{4.10}$$

Equation 4.9 uses the absolute TD-error δ_i directly as the probability of experience i being sampled. The small constant ϵ prevents probabilities of 0. However, this is sensible to outliers, where experiences with very high TD-errors dominate the sampling process. Equation 4.10 mitigates this by using the reciprocal rank of the experiences sorted by the magnitude of the TD-error in descending order. As we have briefly mentioned in Section 2.5.1, if the distribution of the updates does not match the distribution of its expectation, some form of importance sampling is needed to correct the bias. PER therefore scales the TD-errors in the gradients by weights that are proportional to the corresponding probability of each sample. We will not discuss this in further detail, since we do not apply prioritized importance sampling in the practical part of this thesis, as discussed in Section 7.1. As with the other improvements covered before, for the majority of Atari games, PER yielded improved performance.

4.2.4 Rainbow

All of the previously discussed improvements to DQN, and others that we chose to omit, tackle varying issues with regular DQN, but are built on top of a common framework. Therefore, they are complementary and can be combined. This variant of DQN was dubbed "Rainbow", and led to massive improvements on the Atari benchmark [40]. Rainbow DQN also employed distributional RL, which is a topic of its own and will be discussed in the next section.

4.3 Distributional Reinforcement Learning

So far, the methods from Chapter 2, as well as DQN and its variants, were concerned with maximizing the expected return. However, by focusing on this point estimate, one loses potentially interesting information about the set of experienced rewards. Typically, complex environments will contain elements of randomness, with the rewards obtained for a specific state-action pair having uncertainty and variance to them. Therefore, the mean of the random return might simply be a poor representation of the underlying distribution. For instance, in the case of a bimodal distribution, a state-action pair could sometimes yield a high positive, or a high negative reward. The estimated expected return, i.e. the average, will be around 0, a reward value that was never actually observed. The answer to these flaws is to take a distributional perspective and focus on the entire return distribution, in order to capture the intrinsic randomness of the rewards and

environment dynamics. To that end, a full distributional Bellman equation is used:

$$Z_\pi(s,a) \overset{\mathcal{D}}{=} R(s,a) + \gamma Z_\pi(S',A') \tag{4.11}$$

As opposed to the original Bellman equations that related scalars, the distributional one relates random variables, namely the reward R, the next state-action pair (S',A') and the random return $Z_\pi(S',A')$ obtained from that point onward, following policy π. $Z_1 \overset{\mathcal{D}}{=} Z_2$ denotes that the random variables Z_1 and Z_2 are equal in distribution, i.e. $\mathcal{D}(Z_1) = \mathcal{D}(Z_2)$. The familiar action-value is then the expected value of the value distribution Z_π:

$$q_\pi(s,a) \doteq \mathbb{E}[Z_\pi(s,a)] \tag{4.12}$$

The value function maps state-action pairs to values, the value distribution maps them to distributions. While this distributional perspective is not a novel idea [45], it had little impact on RL. Only in the last years, several algorithms have taken this approach, leading to many state-of-the-art results. We will discuss some of them in this section, starting with the Categorical DQN presented by Bellemare et al. in [6], a seminal paper that gave rise to the category of distributional RL methods, and was used in the aforementioned Rainbow DQN. We also rely on [95] and [7] as sources on distributional RL.

4.3.1 Categorical DQN: C51

An important concept presented by the authors is the notion of a Bellman operator $\mathcal{T}^\pi$ and an optimality operator $\mathcal{T}$:

$$\mathcal{T}_\pi Q(s,a) \doteq \mathbb{E}[R(s,a)] + \gamma \mathbb{E}_{P,\pi}[Q(s',a')] \tag{4.13}$$

$$\mathcal{T} Q(s,a) \doteq \mathbb{E}[R(s,a)] + \gamma \mathbb{E}_{P}[\max_{a'} Q(s',a')] \tag{4.14}$$

Iterative application of these operators to an initial Q_0 will converge to Q_π or Q_*. Moreover, the operators are γ-contraction mappings. This means that the same operator applied to two different value functions Q_1 and Q_2 will reduce the distance between them by a factor of at least γ. In other words, this property formalizes that each value function that the operator is applied on, moves closer to the target value, i.e. closer to each other. Mathematically, a γ-contraction mapping is expressed as follows:

$$\text{dist}(\mathcal{T}Q_1, \mathcal{T}Q_2) \leq \gamma \, \text{dist}(Q_1, Q_2) \tag{4.15}$$

Clearly, these operators simply describe the expected behavior of Q-learning. The corresponding Bellman operator $\mathcal{T}_\pi$ for Z_π is defined as:

$$\mathcal{T}_\pi Z(s,a) \overset{\dot{\mathcal{D}}}{=} R(s,a) + \gamma P_\pi Z(S',A') \tag{4.16}$$

P_π is the transition operator that encapsulates the environment dynamics, i.e. the distribution of the next state-value pair (S',A').

$$P_\pi Z(s,a) \overset{\dot{\mathcal{D}}}{=} Z(S',A') \tag{4.17}$$

Without going into detail, which is outside the scope of this thesis, the authors provided proof that using the maximal form of the Wasserstein metric as a distance measure between distributions, the Bellman operator for value distributions is a contraction. However, the same cannot be said for Kullback-Leibler divergence. An operator $\mathcal{T}$ is then said to be a distributional Bellman optimality operator, if the policy π implements greedy action selection. This optimality operator is not a contraction in any distance measure.

The algorithm proposed by the authors is based on the distributional Bellman optimality operator. Its iterative application on an approximated distribution should converge to the set of optimal value distributions. They opted for a categorical distribution as a model, with its support consisting of atoms z_i. The support of a distribution is comprised of all outcomes that can be assigned a nonzero probability. The set of atoms z_i is defined as:

$$\{z_i \quad = \quad V_{min} + i\Delta z \ : \ 0 \leq 1 < N\} \tag{4.18}$$

$$\Delta z \quad \doteq \quad \frac{V_{max} - V_{min}}{N - 1} \tag{4.19}$$

V_{min}, V_{max} and N are the parameters of this distribution, where the two former values limit the action-value range, and the latter value denotes the number of atoms to be used. Usually, $N = 51$, hence the name Categorical 51 (C51). In contrast to DQN, we now use an LDNN to obtain a value distribution Z_θ for a state-action pair (s, a), instead of action-values, where θ again denotes the network's parameters. Using a softmax, the probability of each atom, i.e. $Z_\theta(s, a) = z_i$ for $0 \leq i < N$, is defined as:

$$P(Z_\theta(s, a) = z_i) \quad \doteq \quad p_i(s, a) \quad \doteq \quad \frac{e^{\theta_i(s,a)}}{\sum_j e^{\theta_j(s,a)}} \tag{4.20}$$

Following Equation 4.12, we can then compute the corresponding action-value. These values can be regarded as a weighted ensemble of returns. The idea is that this estimate of the expectation will be more accurate than in DQN, due to the possibility of errors in the probabilities returned by the Z-network (an ANN analogous to the Q-network, but trained to learn the distribution Z instead of Q) canceling each other out [66].

$$\begin{aligned} Q(s, a) \quad &\doteq \quad \mathbb{E}[Z(s, a)] \\ &= \quad \sum_{i=0}^{N-1} z_i p_i(s, a) \end{aligned} \tag{4.21}$$

The problem with a discrete distribution is that applying the operator $\mathcal{T}$ to Z_θ leads to disjoint supports. Furthermore, using the Wasserstein metric as a stand-in for the loss between $\mathcal{T}Z_\theta$ and Z_θ is impossible when learning from experiences. Again, the proof is outside of the scope of this work. The authors solved this issue by projecting the sampled Bellman updates $\hat{\mathcal{T}}Z_\theta$ onto the support of Z_θ. Given an experience tuple $e = (s, a, r, s')$, the Bellman update for each atom z_i is defined as

$$\hat{\mathcal{T}}z_i \quad \doteq \quad r + \gamma z_i \tag{4.22}$$

The projection Φ is then done as follows. Each misaligned atom z_j is split, by distributing its probability $p_j(s', a')$, where a' is the result of greedy action selection with Equation 4.21, to its two neighboring atoms. The probability is weighted indirectly proportionally to the distance to each respective neighbor. For each i, the distributed probabilities are accumulated into the projected Bellman update:

$$(\Phi\hat{\mathcal{T}}Z_{\theta^-}(s,a))_i \quad = \quad \sum_{j=0}^{N-1}\Big[1-\frac{|[\hat{\mathcal{T}}z_j]_{V_{min}}^{V_{max}}-z_i|}{\Delta z}\Big]_0^1 p_j(s',a') \tag{4.23}$$

Note that the parameters θ^- denote the use of a target network, which we have introduced as a key part of DQN in Section 4.2. The four steps of the projected distributional Bellman update are shown in Figure 4.1, along with a detailed visualization for the distribution of probabilities to neighboring atoms.

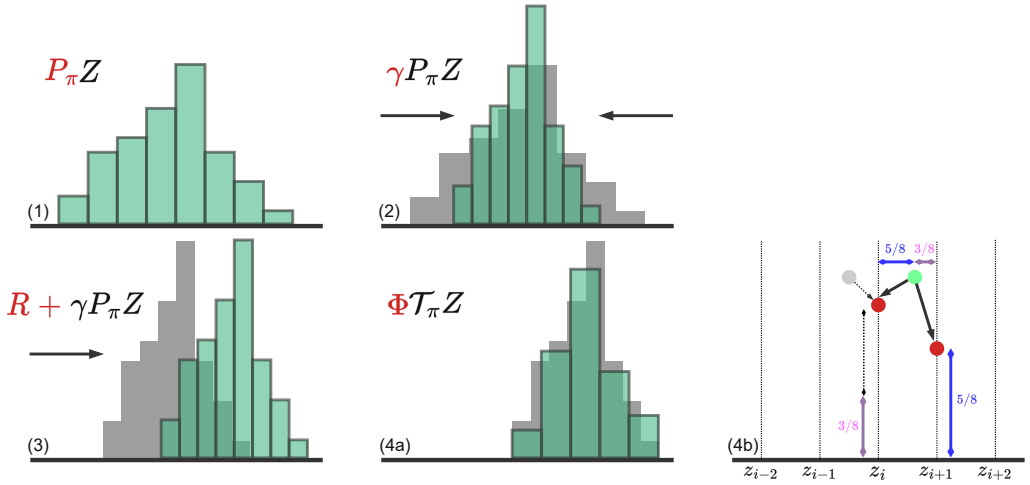


Figure 4.1: The projected distributional Bellman update used in C51, broken down into four steps. (1) Applying the transition operator P_π to the value distribution Z yields the distribution of the next state-value pair. (2) Multiplying with the discount factor γ shrinks the domain, which entails increased probabilities of the support values. (3) Adding the reward shifts the support to the right or left, depending on the reward's sign. (4a) The projection of the Bellman update onto the support distributes the probability of misaligned atoms to its immediate neighbors. (4b) The weight assigned to neighboring atoms is indirectly proportional to the distance from the misaligned atom. All contributions from misaligned atoms are summed.

Ultimately, the projection step permits the use of KL divergence between categorical

distributions as a loss function:

$$\begin{aligned}\mathcal{L}_i(\theta_i) &\doteq D_{KL}(\Phi\hat{\mathcal{T}}Z_{\theta_i^-}(s,a) \,||\, Z_{\theta_i}(s,a)) \\ &= D_{KL}(m_i \,||\, p_i) \\ &= \sum_{i=0}^{N-1} m_i \log m_i - \sum_{i=0}^{N-1} m_i \log p_i \\ &= -H(m) + H(m,p) \end{aligned} \tag{4.24}$$

$$\begin{aligned}\nabla_{\theta_i}\mathcal{L}_i(\theta_i) &= \nabla_{\theta_i} H(m,p) \\ &= -\sum_{i=0}^{N-1} m_i \log p_i(s,a;\,\theta) \end{aligned} \tag{4.25}$$

Since the probabilities m_i come from the target network and do not depend on θ, we can simply use the cross-entropy loss for minimization.

The authors report that C51 was able to outperform DQN and its improvements on the majority of Atari games, obtaining significantly higher mean and median scores. Notably, the agent improved on games with sparse rewards, which indicates that value distributions are better at propagating rare events.

4.3.2 Quantile Regression DQN: QR-DQN

The algorithm we will discuss in this subsection was built on top of the theoretical results presented in the previous subsection. The authors of C51 were able to show that the Wasserstein metric would be an ideal stand-in for the loss. However, since it generally cannot be minimized via gradient descent, C51 runs a projection and subsequently minimizes the KL-divergence between the predicted distribution and the projected Bellman update. This left the theoretical results unused, and whether a distributional algorithm exists that utilizes the Wasserstein metric remained an open question. The work by Dabney et al. [19] closes this gap by employing the theory of quantile regression [47], leading to the algorithm Quantile Regression DQN (QR-DQN).

First and foremost, the authors transposed the fundamental distribution model of the predecessor algorithm. C51 learned to attach probabilities p_i to atoms with fixed locations z_i, for $0 \leq i < N$. In contrast, QR-DQN uses fixed probabilities, namely uniform weights $p_i = 1/N$ for $1 \leq i \leq N$, with variable positions. So the goal is to estimate quantiles of the target distribution, hence called a quantile distribution. The cumulative probabilities τ_i, i.e. the values of the cumulative distribution function, are the cumulative sum of the probabilities p_i, meaning $\tau_i = \frac{i}{N}$ (we assume $\tau_0 = 0$). Mathematically, the Z-network aims to find a quantile distribution Z_θ, which maps a given state-action pair (s,a) to a uniform probability supported on $\theta_i(s,a)$:

$$Z_\theta(s,a) \doteq \frac{1}{N}\sum_{i=1}^{N} \delta_{\theta_i(s,a)} \tag{4.26}$$

The term δ_z is the Dirac, i.e. a unit impulse, at $z \in \mathbb{R}$. In comparison to C51, this distribution is not restricted by boundaries V_{min} and V_{max}, which required domain knowledge, or uniformly spaced atoms. Furthermore, disjoint supports are not an issue. Since the Wasserstein metric operates on quantile functions, i.e. inverse CDFs, the goal of QR-DQN is to find the supports θ_i for $1 \leq i \leq N$ from 4.26 that minimize the metric with respect to the target distribution Z. The authors show that these supports can be obtained by plugging the quantile midpoints $\hat{\tau}_i = \frac{\tau_{i-1}+\tau_i}{2}$ into the quantile function of Z:

$$\theta_i \quad = \quad F_Z^{-1}(\hat{\tau}_i) \tag{4.27}$$

In other words, QR-DQN aims to learn the quantile function, for which it uses quantile regression.

Note that a quantile $\tau \in [0, 1]$ simply denotes that the probability mass left of it is τ, and right of it is $1 - \tau$. For instance, the median is $\tau = 0.5$, because it splits the distribution's probability mass in half. If we want to determine a specific quantile, we must start with an estimate θ and improve it by sampling from the distribution, applying weights $1 - \tau$ and τ to samples below and above θ respectively. Once our estimate is correct, sampling m times from the distribution should yield τ samples below and $1 - \tau$ samples above the estimate. If that is the case, applying the aforementioned weights will lead to an equivalency $(\tau m)(1 - \tau) = ((1 - \tau)m)\tau$. The weights to be applied can be expressed as follows, with $\hat{Z}$ denoting a sample from Z:

$$|\tau - \delta_{\hat{Z}<\theta}| = \begin{cases} |\tau - 1| = 1 - \tau & \text{if } \hat{Z} < \theta \\ \tau, & \text{if } \hat{Z} \geq \theta \end{cases} \tag{4.28}$$

The so called quantile regression loss is then the expected value of multiplying the absolute difference between the sample and the estimate with the corresponding weight:

$$\begin{aligned} \mathcal{L}_{QR}^{\tau} &\doteq \mathbb{E}_{\hat{Z}\sim Z}[|\hat{Z} - \theta||\tau - \delta_{\hat{Z}<\theta}|] \\ &= \mathbb{E}_{\hat{Z}\sim Z}[\rho_\tau(\hat{Z} - \theta)], \qquad \text{with} \qquad (4.29) \\ \rho_\tau(u) &= u(\tau - \delta_{u<0}) \qquad (4.30) \end{aligned}$$

Note that in the last line, we can omit the absolute values, because both factors will always have the same sign.

In summary, given a distribution Z and a quantile τ, the values that yield the minimal quantile regression loss are $F_Z^{-1}(\tau)$. Due to Equation 4.27, the supports we want to learn minimize the sum over all respective quantile regression losses for the quantile midpoints $\hat{\tau}_i$, i.e.:

$$\sum_i^N \mathbb{E}_{\hat{Z}\sim Z}[\rho_{\hat{\tau}_i}(\hat{Z} - \theta_i)] \tag{4.31}$$

This loss can be minimized via stochastic gradient descent, which was the original issue with C51. Last but not least, the authors argue that the loss function's cusp at 0 hinders

optimization. In order to mitigate that they propose the quantile Huber loss, which uses the Huber loss $\mathcal{L}_\kappa$ with parameter κ [44]:

$$\rho_\tau^\kappa(u) = |\tau - \delta_{u<0}|\,\mathcal{L}_\kappa(u) \tag{4.32}$$

$$\mathcal{L}_\kappa(u) = \begin{cases} \frac{1}{2}u^2, & \text{if} |u| \leq \kappa \\ \kappa(|u| - \frac{1}{2}\kappa), & \text{otherwise} \end{cases} \tag{4.33}$$

This turns the quantile regres uared loss within the interval $[-\kappa, \kappa]$. Figure 4.2 shows a vi

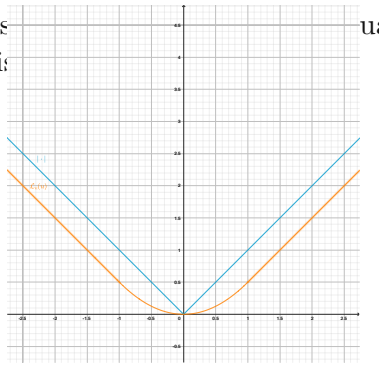


Figure 4.2: Huber loss $\mathcal{L}_\kappa$ for $\kappa = 1$ compared to the absolute value function $|\cdot|$.

Ultimately, QR-DQN uses analogue formulas to Equations 4.21 and 4.22, for action selection and iterative improvement of the quantile estimates, given experience tuples $e = (s, a, r, s')$:

$$Q(s, a) \doteq \sum_i p_i \theta_i(s, a) \tag{4.34}$$

$$\hat{\mathcal{T}}\theta_i \doteq r + \gamma\theta_i(s', a') \tag{4.35}$$

As per usual, QR-DQN outperformed its predecessors on the Atari benchmark. Furthermore, the authors evaluated the added improvement of the quantile Huber loss, with the parameters $\kappa = 0$ (i.e. regular ρ_τ) and $\kappa = 1$, showing empirically that the latter outperforms the former.

4.3.3 Implicit Quantile Networks: IQN

The algorithm presented in this subsection again expands on the theories and algorithms of distributional RL discussed in this section so far. Therefore, we first briefly summarize C51 and QR-DQN. Both aim to learn a parameterized estimate of the return distribution Z. The former attaches priorities to a discrete set of a priori determined support values. The latter transposes this representation, by assuming a uniform mixture of unit impulses and adjusting their respective locations via quantile regression. In contrast to C51, QR-DQN utilizes the Wasserstein metric. It aims to return quantiles that minimize the distance between the target and the estimated distribution, updated with the distributional Bellman update. The next logical extension to QR-DQN is to remove the restriction of discrete sets. Implicit Quantile Networks (IQN) [20] does this by learning the entire quantile function, instead of only a set of discrete quantiles.

IQN denotes not only the algorithm, but specifically the deterministic parametric function that is being trained. It takes as input samples τ from a base distribution, for which the authors propose a uniform distribution $U([0,1])$, and outputs the respective quantile values at τ for the target distribution Z, i.e. $\tau \mapsto F_Z^{-1}(\tau)$. The function can then be used to obtain a sample of the return distribution, i.e. $F_Z^{-1}(\tau)(s,a) \sim Z(s,a)$, for $\tau \sim U([0,1])$. The authors denote the quantile function at τ as Z_τ. The function that learns Z_τ for a given τ is embedded into the LDNN that is used for function approximation. Given a state-action pair (s,a), the quantile function is estimated as:

$$Z_\tau(s,a) \quad \approx \quad f(\phi(s) \odot \psi(\tau))_a \tag{4.36}$$

Concretely, $\phi(s)$ is the part of the LDNN that processes the state, whereas f maps the resulting embedding to action-value estimates. Of course, f can also represent a part of the LDNN consisting of multiple layers. A regular DQN could thus be described as approximating $Q(s,a) \approx f(\phi(s))_a$. IQN adds ψ into the neural network, as an additional arm that maps an input $\tau \sim U([0,1])$ to $\mathbb{R}^d$. The two arms are ultimately combined with an element-wise product denoted by $\odot$. Therefore, the dimension d of the output produced by ψ must match the dimension of the intermediate representation of the state produced by ϕ. While the possibilities to parameterize $\phi(\tau)$ are vast, the authors propose the following definition, with an embedding dimension $n = 64$:

$$\phi_j(\tau) \quad \doteq \quad \text{ReLU}\Big(\sum_{i=0}^{n-1} \cos(\pi i \tau) w_{ij} + b_j\Big) \tag{4.37}$$

In words, the single scalar input τ is expanded to a vector $\mathbf{v} \in \mathbb{R}^i$ with elements $v_i = \cos(\pi i \tau)$ for $i \in [0, n-1]$. $\mathbf{v}$ is then pushed through a fully connected neural network layer, with weights $\mathbf{W}$ and biases $\mathbf{b}$, to obtain an embedding of dimension $\mathbb{R}^d$. Since τ is sampled in every forward pass through the network, we must distinguish between τ and τ'. Given an experience tuple experience tuple $e = (s, a, r, s')$, the TD-error u depends on the two samples $\tau, \tau' \sim U([0,1])$:

$$u_{\tau,\tau'} \quad = \quad r + \gamma Z_{\tau'}(s', a') - Z_\tau(s,a) \tag{4.38}$$

As per usual, s' is the result of greedy action selection. The number of samples τ and τ' to draw, i.e. N and N', can be different for the computation of $Z_{\tau'}$ and Z_τ, done by the target and online network respectively. Just like in QR-DQN, the quantile Huber loss is applied to the TD-error $u_{\tau,\tau'}$ at each step. Finally, the action-value function estimate is defined as the expected value of Z_τ, which can be approximated by sampling $\tilde{\tau} \sim U([0,1])$ K times and taking the average:

$$\begin{aligned} Q(s,a) \quad &\doteq \quad \mathbb{E}_{\tau \sim U([0,1])}\big[Z_\tau(s,a)\big] \\ Q(s,a) \quad &\approx \quad \frac{1}{K} \sum_{k=1}^{K} Z_{\tilde{\tau}_k}(s,a) \end{aligned} \tag{4.39}$$

Generally speaking, the authors found that increasing N and N' improved performance until reaching a plateau. For simplicity, setting $N = N'$ is valid. IQN was found to be insensitive towards the value of K.

4.3.4 Fully Parameterized Quantile Function: FQF

The authors of the IQN algorithm [20] argued that, given enough computational resources for a sufficiently large network, any distribution can be approximated, using an infinite number of quantile fractions in the limit. Of course, in practice, the number of quantile fractions is limited to a comparatively small number of random samples. Again, a logical next step would be to make the quantiles τ dependent on each specific state-action pair, instead of sampling them uniformly at random. To that end, the authors of [94] propose the extension of the IQN model by another neural network, the so called fraction proposal network, giving rise to the Fully Parameterized Quantile Function (FQF) algorithm. The LDNN responsible for approximating the quantile value function, i.e. the Quantile Value Network (QVN), remains unchanged from IQN. But instead of sampling quantiles, they are generated by an additional LDNN, called the Fraction Proposal Network (FPN). In summary, FQF attaches N estimated quantile values to N estimated quantile fractions, with neither being fixed nor sampled. Note that in QR-DQN, the return distribution is approximated with Equation 4.26, for a fixed set of $\tau_i = \frac{i}{N}$ for $1 \leq i \leq N$, and each Dirac is weighted with the term $\frac{1}{N} = \tau_{i+1} - \tau_i$. Analogously, since FQF uses adjustable quantiles, the return distribution is defined as:

$$Z_{\theta,\tau}(s,a) \doteq \sum_{i=0}^{N-1} (\tau_{i+1} - \tau_i)\, \delta_{\theta_i(s,a)} \tag{4.40}$$

In order to leverage the same theoretical results regarding the Wasserstein metric as QR-DQN, which we have briefly discussed in Section 4.3.2, the quantiles constructed by the FPN must have the properties $\tau_{i+1} < \tau_i$, $\tau_0 = 0$ and $\tau_N = 1$. Clearly, training the network to emit values that fulfill these conditions is not trivial. Therefore, the authors propose using a cumulative softmax, i.e. $\tau_i = \sum_{j=0}^{i-1} q_j$ for $i \in [0, N]$, where q_j are softmax output values. The FPN can be trained by minimizing the Wasserstein metric via gradient descent, using its derivative with respect to the proposed τ_i:

$$\frac{\partial W_1}{\partial \tau_i} = 2Z_{\tau_i} - Z_{\hat{\tau}_i} - Z_{\hat{\tau}_{i-1}} \qquad \forall i \in [1, N-1] \tag{4.41}$$

Just like with QR-DQN, $\hat{\tau}_i$ denotes the quantile midpoints. The respective quantile functions are provided from the QVN. Its training process is identical to IQN's, using the quantile Huber loss on the same TD-error (Equation 4.38).

Of course, the training of FQF is slower compared to the distributional RL algorithms discussed so far, due to the addition of another neural network. It is also very sensitive to the number of samples τ, which is not the case for IQN. However, the benefit of a fully parameterized quantile function without fixed components was clearly shown on the Atari games benchmark, with FQF outperforming the other distributional RL algorithms.

Figure 4.3 summarizes and compares the four distributional RL algorithms presented in this section. It also highlights the evolution up to FQF, starting from the non-distributional algorithm DQN.

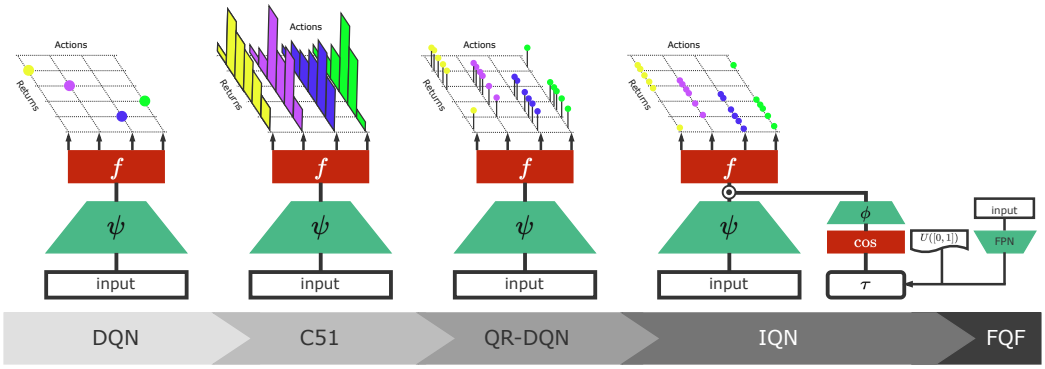


Figure 4.3: Comparison of distributional RL algorithms and DQN. The visualization is an adaptation and extension of a Figure featured in [20]. It shows the adaptations and improvements that led to FQF: DQN yields point estimates, C51 attributes different probabilities to fixed atoms, QR-DQN attaches fixed probabilities to different quantiles, IQN learns the entire quantile function for quantiles sampled from a uniform distribution. Ultimately FQF learns to adapt the quantiles to the input.

4.4 REINFORCE

DQN and its distributional variants fall into the category of value-based methods. These algorithms first learn an estimate of the value-function and then use it to derive a policy, usually one that picks the actions with the highest value. The counterpart to value-based methods are policy-based methods. These algorithms take a more direct approach, by optimizing the policy itself, e.g. by gradient ascent on the objective $J(\theta)$, where θ are the parameters of an ANN representing the policy π. The main goal of these approaches is to update the action probability distribution of a stochastic policy such that actions with high expected reward also have a high probability, and vice versa. A popular policy-based approach is the REINFORCE algorithm. The objective $J(\theta)$ is simply the expected return of an episode, which is maximized via gradient ascent. The policy gradient can be derived using the equation $\frac{d}{dx}\log f(x) = \frac{f'(x)}{f(x)}$, where τ denotes a trajectory:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \sum_{t=i}^{T-1} \nabla_\theta P(s_t, a_t \,|\, \tau) r_{t+1} \\
&= \sum_{t=i}^{T-1} P(s_t, a_t \,|\, \tau) \frac{\nabla_\theta P(s_t, a_t \,|\, \tau)}{P(s_t, a_t \,|\, \tau)} r_{t+1} \\
&= \sum_{t=i}^{T-1} P(s_t, a_t \,|\, \tau) \nabla_\theta \log P(s_t, a_t \,|\, \tau) r_{t+1} \\
&= \mathbb{E}\Big[\sum_{t=i}^{T-1} \nabla_\theta \log P(s_t, a_t \,|\, \tau) r_{t+1} \Big] \qquad (4.42)
\end{aligned}$$

In practice, random past episodes are sampled to estimate the expectation. Briefly speaking, the logarithm simplifies the gradient, because terms that do not depend on θ, such as environment transition probabilities, will be 0. Ultimately, the gradient ends up as:

$$\nabla_\theta J(\theta) \quad = \quad \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t \tag{4.43}$$

REINFORCE collects entire episodes, storing the log-probabilities of the policy, as well as the obtained rewards, at every timestep. The rewards are used to compute the discounted cumulative future reward at each step, which are then plugged into the policy gradient used to update the parameters θ.

4.5 DDPG and TD3

While the DRL algorithms presented so far, i.e. DQN and variants of distributional RL, are able to handle high-dimensional observation spaces, they only work with discrete or low-dimensional action spaces. In the case of continuous actions, e.g. a real-valued actuator setting in a physical control task, finding the specific action with the highest value would require iterative optimization. This process would be too slow to be feasible, especially in the case of large ANNs. Discretizing the action space is usually not an option, since a fine-grained approach can yield an intractably large action space, and a more naive approach leads to the loss of potentially essential information.

A prominent example for a DRL algorithm that can operate in continuous action spaces is Deep Deterministic Policy Gradient (DDPG) [49]. It uses key ideas from the authors of DQN, namely experience replay buffers and target networks, and applies them to Deterministic Policy Gradient (DPG) [79], an off-policy Actor-Critic (AC) algorithm, hence extending the name with the prefix "deep". "Actor-critic" refers to a common architecture used in RL, consisting of an actor that learns a stochastic policy, and a critic that learns the action-value function [83]. In DRL, typically both actor and critic are separate ANNs. The actor is an ANN learning a policy $\pi(s;\, \theta_\pi)$ that deterministically maps an action to a given state s, where θ_π denotes the ANN's parameters. The critic is constructed and trained analogously to DQN, but utilizing the actions determined by the actor. The actor aims to maximize the expected return, via the policy gradient obtained with the chain rule:

$$J \quad \approx \quad \mathbb{E}\big[Q(s, \pi(s;\, \theta_\pi);\, \theta_Q)\big] \tag{4.44}$$

$$\begin{aligned} \nabla_{\theta_\pi} J \quad &\approx \quad \mathbb{E}\big[\nabla_{\theta_\pi} Q(s, \pi(s;\, \theta_\pi);\, \theta_Q)\big] \\ &= \quad \mathbb{E}\big[\nabla_a Q(s, a;\, \theta_Q,\, a = \pi(s;\, \theta_\pi))\, \nabla_{\theta_\pi} \pi(s;\, \theta_\pi)\big] \end{aligned} \tag{4.45}$$

In words, given an experience tuple $e = (s, a, r, s')$, the actor determines the next action a' given s', which is used to update the Q-network, i.e. the critic, as is done in DQN. Subsequently, the critic is used to compute the value of the action produced by the actor for the current state s. Since the goal is to maximize the expected return, the actor's loss

is the negative action-value determined by the critic, i.e. the critic criticizes the actor's choices.

In contrast to DQN, DDPG utilizes so called "soft" target updates to update the target networks. As opposed to directly copying the weights from the online network to the target network every n steps, a soft update occurs at every step. It slowly nudges the target network's parameters towards those of the corresponding online network, according to the following update formula:

$$\theta' \quad \leftarrow \quad \tau\theta + (1-\tau)\theta' \tag{4.46}$$

The parameter τ is typically very small, such as 0.01. It should be briefly noted that in continuous action spaces, exploration is done by simply adding noise to the actor's policy. However, as we will discuss in Section 7.1.5, exploration does not concern us in the practical part of this thesis.

A common issue that can occur with DDPG is the overestimation of action-values by the critic, which subsequently allows the actor to outplay the critic, by producing actions that exploit the critic's weaknesses. In turn, this exploitation leads to the critic returning even higher action-values, which ultimately causes the learning process to spiral out of control. Twin Delayed DDPG (TD3) [21] addresses these issues with 3 adaptations to DDPG. Firstly, similarly to double learning in DQN, TD3 adds an additional critic and uses the smaller action-value of the two when computing the target. This hinders the actor from exploiting the critic, because it has to exploit both of them. Secondly, the target update for the actor network is delayed, with the authors recommending one update to the actor for every two updates of the critic. Thirdly, noise is added to the actor's choice of a', in order to further decrease the chance of exploitation.

A noteworthy algorithm built on top of DDPG/TD3 is the Wolpertinger algorithm [25]. It was designed for the purpose of handling large discrete action spaces, which can occur in language models, industrial plants, or recommender systems with large amounts of items. While applying DQN is certainly possible, it is inefficient. The Wolpertinger algorithm requires that each action is represented by an embedding. The actor produces a so called prototype-action $\hat{a}$, i.e. a vector that corresponds to the dimensions of an action embedding, but does not necessarily represent a valid action from the discrete set of actions $\mathcal{A}$. Given a prototype-action, the k nearest neighbors from the set of valid actions are selected and passed through the critic, which ultimately selects the action with the highest action-value. TD3 can be adapted to have one critic select the action and the other critic return the value for that action. While the critic is trained using the valid actions, the policy gradient is taken at the actor's prototype-action.

CHAPTER 5

NRSs and DRLRSs: State-of-the-Art

In this chapter, we will separately examine the topics News Recommender System (NRS) and Deep Reinforcement Learning Recommender System (DRLRS). We will begin by introducing the news recommendation problem and what differentiates news from other domains. Then, we will present State-Of-The-Art (SOTA) solutions. Besides the literature on the aforementioned solutions, we also rely on the surveys on the topic of NRS research, namely [92], [57], [68], [27], and [46]. Finally, we will examine how DRL can be applied to RSs, and discuss the state of the field of research surrounding this idea, sourcing from [2] and [17].

5.1 Introduction to News Recommendation

The news industry has been subjected to a significant paradigm shift over the past decades, with an increasing amount of people looking to read news online. Users can turn directly to websites of news organizations, social media networks, or news aggregators, such as Google News[1], Apple News[2], or Microsoft News[3]. Nowadays, the majority of U.S. Americans consume news online, via digital devices (e.g. smartphone, tablet, or PC), as opposed to reading print publications, listening to the radio, or even watching TV [14]. Current statistics published by the European Union paint the same picture [26]. The consequence of this was, as we have already discussed in Section 3.1, the rise of today's digital media landscape, characterized by a seemingly infinite supply of news and an accompanying abundance of choice. At the same time, according to the 2022 Digital News Report by the Reuters Institute for the Study of Journalism [62], the growth of

[1] https://news.google.com/
[2] https://www.apple.com/apple-news/
[3] https://microsoftnews.msn.com/

people willing to pay for online news has seemingly stagnated at a low level, and overall interest in news has declined sharply across markets over the past years, to only 51% in 2022. NRSs present a possible solution to these issues. They can navigate the scarce attention of users through the abundance of news content to the information that is particularly interesting or important to them. Subsequently, relieving this information overload can potentially lead to users having increased engagement and loyalty towards news organizations, thus being more willing to pay [10].

NRSs fall into the general framework that we have outlined in Section 3.1, with items being news content in various forms (e.g. articles, videos, or podcasts), and transactions being media-dependent interactions between users and items (i.e. reading, watching, or listening to news content). Feedback is typically implicit, in the form of clicks or views, but could theoretically be explicit as well, if users are prompted to rate items in some way. For instance, users could assess the quality of writing, or how interesting an article was to them. However, compared to domains like movies, advertising, music or e-commerce, the news domain entails a unique set of challenges that makes the development of NRSs a difficult task. Some of these are:

- *Engagement and Survival Time*: Depending on the length of an article, the average engagement time, i.e. the duration for which a user interacts with the content, spans from 43 to 270 seconds (0.7 to 4.5 minutes). Furthermore, the majority of news items has a very short survival time, meaning the content becomes quickly out-of-date, e.g. in a matter of days, hours, or even minutes in the case of developing stories. In contrast, movie items usually have a consumption time measured in hours, and their eligibility for recommendation is not as strongly tied to their age. The same can be said for books and music, where the engagement time of the latter is comparable to news, but the survival time is much longer.

- *Candidate Set Size*: Due to the size of the media landscape discussed before, the release rate of news content is extremely high, in the range of thousands of new items per hour. This leads to a very large set of candidate news items that can be recommended. While this is also the case in other domains, e.g. music or e-commerce, the addition of a low survival time in the news domain means that the candidate set is constantly changing.

- *Sequential Consumption*: While the order of consumption is an important factor in all RSs, it is paramount in the news domain. For example, an RS might correctly identify a user's interest in a breaking story, and recommend a corresponding news item from a set of candidate news that cover the topic. The user's engagement with the suggested item can then lead to a decrease, or even complete end, of their interest. In another case, a user could want to remain informed on a developing story, thus requiring recommendations for news items containing new information in the order of discovery. Last but not least, unlike in the music domain, items in the sequence of consumption are usually not repeated.

- *Context*: Situational and contextual aspects can have a significant influence on user's preferences and interests. Current events and trends can encourage an otherwise uninterested user to consume corresponding news content. For instance, a quadrennial event such as the Olympics, or the world's best football player unexpectedly leaving their childhood club, can motivate typically sports-averse users to interact with items covering these events. Time is also an important factor, where the time-of-day was found [54] to affect the preferences for certain news categories. Concretely, the political category was generally preferred in the morning, whereas the entertainment category was more popular in the evening. Another key context is a user's location, which can naturally lead the user to engage with local news that usually do not pertain to them. Finally, some contextual properties can be highly personal, such as a user's mood or the current weather affecting their preferences. To summarize, users in the news domain have highly dynamic preferences that an RS constantly has to adapt to.

As already said, these properties present issues that are either non-existent, or less pronounced in other domains. In addition, one can easily argue that NRSs have a unique status, due to their potential impact on journalism as an industry, as well as the public's information consumption behavior, which ultimately can affect democratic processes and systems as a whole. In other words, the recommendation of news content has a greater social relevance than the recommendation of movies, music, books, or similar types of items. We briefly summarize the main arguments for the aforementioned elevated status of NRSs:

- *News Supply*: An increasing utilization of NRSs could lead to a fully automated curation and dissemination of news content, free of human editors. [12] argues that this could shift the focus in the processes of journalistic production and selection away from topics that deserve attention, towards topics that will be recommended. For instance, sensationalism could thrive. This effect would naturally be exacerbated by the direct insight into the user's demand, via metrics supplied by the NRSs, thus changing the perception of what designates qualitative journalism.

- *News Demand*: The inherent goal of an RS, i.e. to elicit users' preferences and suggest matching items, can have negative effects on the quality of the users' overall "dietary intake" of news. The assumption that an RS understands what a user wants can lead to filter bubbles, i.e. users being isolated in regards to the spectrum of items they are able to transact with. Especially in the category of politics, engaging with counter-attitudinal opinions is important. News media are central institutions that should ensure that the citizenry remains informed and attends to a broad and diverse set of opinions and ideas. Failing to do so could have negative implications for democracy [39]. However, people tend to selectively choose items that are attitude-consistent, meaning aligned with their beliefs and viewpoints, and avoid attitude-dissonant content. While the former can reinforce confidence

into their pre-existent perspective, the latter can cause an increase in psychological distress and uncertainty [5]. Therefore, an NRS could theoretically aim to serve users with exclusively attitude-consistent items, thus trapping them in an echo chamber [60].

While these issues are being framed as challenges, they present an opportunity for positive impact as well. Clearly, the development of commercial NRSs must include these beyond-accuracy aspects not only in the evaluation, but also directly in the engineering process. However, this is a highly complex topic of its own, and thus not in the scope of this thesis.

In comparison to other domains, the recommendation problem for news content has been studied significantly less extensively. A key reason could be the scarcity of public datasets for offline training and evaluation [93]. The majority of existing datasets are private, or synthetically developed for a narrow research purpose. However, all surveys cited at the beginning of this chapter have found an increase in research output over the past decade. The CLEF NEWSREEL challenge, an annual research competition on the topic of NRSs that ran from 2014-2017, has played an important part in this rise. Participants were given access to a comprehensive dataset and an existing NRS, which encouraged research and led to a spike in the number of publications during the challenge's active years. Furthermore, a public, large-scale dataset for news recommendation, called **MI**crosoft **N**ews **D**ata (MIND), was released in 2020, along with an accompanying challenge, thus fostering additional research and combating the aforementioned scarcity [93]. It is the principal dataset used in the practical part of this thesis, and will be discussed in Chapter 6.

5.2 State-of-the-Art: News Recommender Systems

All of the surveys cited at the beginning of this chapter have identified that, out of the general recommendation methods presented in Section 3.2, collaborative filtering is applied the least in the news recommendation domain. Generally, either content-based or hybrid approaches are the most used methods overall, depending on which papers were analyzed, and the total amount of included papers. Furthermore, [68] conclude that there has been an increased utilization of DL methods for RSs in the past years, which can be observed in the news domain as well. In fact, all of the SOTA methods that we will discuss in this section use DL techniques and architectures. Of course, a key aspect of DL is the ability of a model to generalize. For instance, let's assume a model is given information about a user's past activities on a NRS platform, e.g. consisting of a list of items the user has interacted with, and learns to predict the probability of that user clicking on a new item. Due to the nature of DL, where the parameters of a model are adapted with each new observation, the model will automatically apply what it learned from one user's transactional data to other users. Therefore, while we have described a content-based approach, one can argue that DL methods implicitly contain a

collaborative filtering component, thus placing them in the category of hybrid techniques. Unlike in traditional collaborative filtering, DL finds similarities between users in a latent space, instead of directly comparing the feedback users have given to items both have transacted with. While the other surveys do not discuss knowledge-based methods, [46] points out that none of the analyzed papers utilize techniques from that category. The authors state that this is not surprising, since the news domain does not fit the typical use-case of knowledge-based methods, which we have also discussed in Section 3.2. To reiterate, these techniques are usually applied in settings comprised of complex items with long life-cycles and few, but consequential, transactions. The news domain is essentially the exact opposite of this description, as we have extensively laid out in the preceding section. However, we offer potential applications of knowledge-based techniques in news recommendation in Section 8.3.

Due to the previously discussed scarcity of public datasets, leading to the reliance on proprietary, private data, results in NRS research were often not comparable or reproducible, especially after the CLEF NEWSREEL challenge was discontinued. However, the aforementioned release of the MIND dataset has facilitated both. Therefore, we will briefly present recommendation methods that have performed well when applied to the MIND dataset. Note that for most of these methods, their respective authors report that ensembles of several, independently trained models have achieved the best evaluation scores. Most notably, the current highscore on the MIND evaluation leaderboard[4] was achieved by an ensemble model. We think that it is safe to assume, that these methods represent the SOTA for the entire NRS domain, as opposed to only for this specific dataset. Nonetheless, note that these methods are focused on performing well on accuracy-based measures, which is the evaluation approach implicitly enforced by the authors of the MIND dataset, because their testbed ranks solutions based on the obtained AUC score. The solutions contributed in the practical part of this thesis (see Chapter 7) are thus also accuracy-centric recommender systems. In summary, we do not consider beyond-accuracy evaluations in this section, but want to again emphasize their overall importance, especially in the light of the arguments put forward in the previous section, about the societal importance of NRSs.

5.2.1 Neural News Recommendation with Long- and Short-term User Representations: LSTUR

An et al. proposed a neural, i.e. DL, approach to news recommendation in [4]. Some of the authors were part of the team behind the MIND dataset release. Their proposal consists of two key components, namely a news encoder and a user encoder. The news encoder converts textual attributes of a news article into sequences of word embeddings. The matrix obtained by stacking these embeddings is subsequently passed through a Convolutional Neural Network (CNN). The CNN learns to extract contextual word embeddings, by sliding a window of size M over the entire sequence. Clearly, the context around a word is highly important to its meaning. For instance, in the article

[4] https://msnews.github.io/#leaderboard

title "Houston Rockets End 2022-23 Season In Win vs. Wizards"[5] various word-level interactions are paramount for understanding the context of this news item. Representing this item as a sequence of regular word embeddings would presumably not accurately encode the information that "Rockets" and "Wizards" are names of NBA teams, and the term "Season" thus is a sports-related portion of the year, as opposed to being climate-related. The resulting embeddings are then further processed by an additive word attention network. Its goal is to produce encodings that focus on important words, via attention weighted summation. Finally, the resulting vector is concatenated with embeddings of the category and subcategory of the respective news item. The user encoder uses the list of previously clicked news of a user to build a model for that user. It itself consists of two separate modules, namely a short-term and a long-term user representation (STUR/LTUR) model. After each news in a user's history is passed through the news encoder, the entire sequence is processed by a Gated Recurrent Unit (Network) (GRU). The short-term representation vector is then the GRU's last hidden state. The long-term user model attempts to capture the general preferences of a user, by embedding the user's randomly initialized ID. The authors propose two approaches to combining the short- and long-term encodings. Firstly, using the long-term vector to initialize the GRU's hidden state. Secondly, concatenating the two encodings. The authors report slightly better results for the former method. Furthermore, they show that it outperforms a model that solely relies on the STUR. Unsurprisingly, only using the LTUR in a model yields bad results, since it completely disregards the transaction histories. In addition, the authors also experimented with various methods for learning short-term user representations. Besides GRU, they tested Long Short-Term Memory (LSTM), weighted summation via additive attention and simple averaging over all items. The last two were outperformed by the sequential methods. Perhaps unexpectedly, GRU led to better results than LSTM. The authors put forward the possible explanation that due to GRU having fewer parameters, it is less likely to overfit. In the news encoder, using an LSTM, with or without attention, also was reportedly outperformed by the CNN approach (with and without attention). Last but not least, the authors verified the added benefit of including embeddings of a news item's category and subcategory, finding that using both was the best choice. On its own, the subcategory led to a larger improvement than the category. Finally, given a candidate news article, the probability of a user clicking on it is computed as the inner product between the user encoding and the encoding of the candidate item, produced by the news encoder. A set of candidate items can then be ranked by their respective click probability for a specific user in decreasing order

5.2.2 Neural News Recommendation with Multi-Head Self-Attention: NRMS

Wu et. al, half of whom also worked on the previously presented model, and many on the MIND dataset, improved on LSTUR with NRMS [91]. Again, the model consists of a

[5] https://www.si.com/nba/rockets/news/houston-rockets-end-2022-23-season-win-vs-washington-wizards

news encoder and a user encoder. The news encoder also first converts textual attributes of a news article into sequences of word embeddings. However, instead of using a CNN, these sequences are passed through a multi-head self-attention network. In contrast to a CNN, a self-attention mechanism is able to capture even long-distance interactions and contexts of words. For instance, in the previously given headline, a CNN will presumably understand that "Rockets" refers to an NBA team, given the prefix "Houston". But it might miss the interaction between "Rockets" and "Wizards", because the two words are relatively far apart. Finally, the sequence of multi-head word representations, obtained by concatenating the encodings produced by each head, is passed through an additive word attention network. Again, the user encoder uses the list of previously clicked news of a user to build a model for that user. After each news in a user's history is passed through the news encoder, a multi-head self-attention network is applied again. In this case, it focuses on interactions between news items in a history, with the aim of learning how the items relate to each other. Subsequently, analogously to the news encoder, an additive news attention network learns to differentiate between more and less important items in a history. Again, attention weighted summation ultimately yields a single user representation vector. To compute a click probability score for a given candidate item, the authors again utilize the dot product. The authors show the effectiveness of the different attention networks and levels, by comparing the performances of the model, when individual attention mechanisms are excluded. Word-level attention was shown to be more effective than news-level attention, and self-attention was more effective than additive attention. However, in both cases, using both led to the best performance. In a later work [88], the authors explore the utilization of a large Pre-Trained Language Model (PLM) in the news encoder component. Due to their depth and overall size, as well as the typically extensive pre-training on large corpora of unlabeled text, PLMs present a better starting point, than the comparatively shallow text modeling employed otherwise. In addition, a PLM can be fine-tuned during training, if desired. The authors report significant improvements across the board, when enhancing the regular news encoder in NRMS with a PLM. Furthermore, the authors tested different approaches to pooling the hidden states of a PLM, in order to produce a final embedding for the input news item. Using an attention network performed better than averaging over all hidden states, which in turn bested the widely-used approach of directly using the PLM's output for the "[CLS]" token, or whichever other token is designed to represent the entire sequence.

5.2.3 User-News Matching BERT for News Recommendation: UNBERT

The two architectures presented so far have relied on separate news and user encoders, where the former learns to represent news items as vectors, and the latter applies the former to the items in a user's list of previously read articles, in order to learn a representation vector for said user. Given a user and their history, along with a candidate news item, the separately obtained encodings are then used in later stages of the model to generate a final matching score, or click probability score. However, the authors of [96]

argue that these two-pronged approaches potentially ignore matching signals occurring on a lower level, such as relations on the word-level. To that end, they propose a joint approach, where the textual content of the candidate item and the items in a user's history are concatenated, and subsequently processed together. Just like we discussed at the end of the previous subsection, the authors also leveraged the power of a PLM, concretely BERT [22]. The input to the BERT model was constructed by summing individual, pre-trained embeddings on the level of tokens, segments, positions and news items. In other words, the input representation contains information that separates the candidate item from the user history, each respective item, as well as the individual tokens. The architecture is split into a word- and a news-level module. The first module learns an encoding for each word using multiple transformer layers, along with an overall word-level matching vector, which is supposed to encode how well the candidate item matches the user's preferences on a word-level. This vector was the final output for BERT's "[CLS]" token. The second module then aggregates the word encodings for each corresponding news item, and passes the results again through several transformer layers, in order to capture matching signals on the news level. Again, the resulting output for the "[CLS]" token represents a matching vector, this time on the news-level. The authors also tested the same three approaches for aggregating the word-encodings that we discussed in the previous subsection, namely attention-pooling, mean-pooling, and using the embedding of a special separator token (in this case, they used the token that separated news items from each other in the input). Analogously, attention outperformed averaging, which outperformed just using the token embedding. Ultimately, the word-level and the news-level matching vectors are concatenated and fed into a fully-connected layer, producing a click probability score. Similarly to the NRMS, the authors also put forward an ablation study, where they disable each of the two modules individually, to test its impact on the overall performance. They report the best results when both modules are used. Unsurprisingly, in the case of a single module, the news-level module outperforms the word-level module.

5.2.4 Multi-Interest Matching Network for News Recommendation: MINER

While other news recommendation architectures typically learn to represent users with a single embedding vector, the authors of [48] propose a poly attention mechanism. They argue that due to the diversity and variance inherent to users' interests, relying on a single-vector encoding presents a bottleneck. To relax this degree of compression, they propose using K learnable context codes, where each code influences the attention mechanism applied to the news items in a user's history. The result is K user model vectors, where in theory, each vector represents a different aspect of their preferences. In addition, the authors introduce a regularization that aims to minimize the mean cosine similarity between the individual representation vectors. The goal is to ensure that each pairwise distance is high and vectors are dissimilar, i.e. a diverse set of vectors, which the authors call disagreement regularization. Finally, for a given candidate news

item, a matching score is computed for each of the K vectors, as the dot product with the encoding of a candidate news article. Out of three proposed ways to aggregate the resulting K scores, namely taking the maximum, the mean, and an attention-weighted sum of each score, the last approach outperformed the others. The authors reportedly obtained best results for $K = 32$. Last but not least, MINER re-weighs news items in a user's history, proportionally to the similarity to the candidate item, in terms of its news category. For instance, if the candidate item is a sports news article, the items in the user's history from the categories sports, fitness, health and food would receive a higher weight. Following [88], MINER uses a PLM to enhance the news encoder component. Again, the authors conduct an ablation study, proving the effectiveness of individual MINER components (disagreement regularization, category weighting and the PLM), by obtaining worse performances when removing them.

5.2.5 Fastformer: Additive Attention Can Be All You Need

Finally, the currently best result on the MIND leaderboard[6] was achieved by a model called Fastformer [89], developed by the same authors as NRMS. The self-attention mechanism in the standard issue transformer has to compute the pair-wise dot product for the entire sequence of input vectors, meaning its complexity is quadratic to the length of the sequence. Therefore, longer sequences are not handled efficiently. To that end, the authors propose the Fastformer, which heavily utilizes additive attention to achieve linear complexity. It first uses additive attention to compress the input query matrix into a single global query vector. The interaction with the keys in the input key matrix is then modeled via element-wise multiplication with the global query vector. The resulting matrix is again compressed into a global key vector with additive attention. Ultimately, element-wise multiplication of the global key vector and the input attention values, along with a subsequent linear transformation of the resulting vectors, yields global attention values. The sum of these values with the original query vectors forms the final Fastformer output. The authors utilize Fastformer as a news encoder, by feeding it with GloVe embeddings as the initial token embedding matrix, and using additive attention to combine the output vectors into a single news encoding vector. Analogously to previous works, the news encoder is applied to the sequence of previously read news items to produce a user encoding. While the authors do not specify how the individual item encodings are combined into a user encoding, one can assume that they reused the combination of self- and additive attention that was used in NRMS. Following [88], the authors report improved results when combining Fastformer with a PLM. Furthermore, using a Fastformer that was pre-trained on unspecified data also boosts the model's performance. Last but not least, the authors tested the Fastformer architecture on additional tasks besides news recommendation, namely sentiment and topic classification, as well as text summarization. Fastformer not only achieved significantly lower training and inference times than other transformer variants with linear complexity, it also outperformed many of them on the tested tasks.

[6] https://msnews.github.io/#leaderboard

5.3 (Deep) Reinforcement Learning Recommender Systems

The RSs discussed so far have viewed item recommendation as a prediction or classification problem, e.g. by aiming to predict a user's rating of an item, a click probability, or to classify items into interesting and uninteresting groups. However, the interactions between users and items within an RS are sequential processes, which suggests that the recommendation problem can be viewed as a sequential decision problem [78]. Consequently, modeling recommender systems as MDPs and applying RL algorithms is a viable option. RL has the potential to handle the dynamic nature of user preferences in an RS better than conventional methods, by continuously adapting to the received reward, whereas periodic re-training is required otherwise. At the same time, an RL agent could accurately take into account user engagement and interest in the long-term, due to the agent's aim to maximize return. While tabular RL methods are not feasible in most RSs, due to the sizes of the item and user spaces, using DL for function approximation solves this issue, and the DRL methods discussed in Chapter 4 are indeed applicable. In summary, DRLRSs are a promising research direction. [2] reports that in the years following the publication of the seminal paper on DQN [58], the amount of publications on the topic of DRLRSs has steeply increased, and the application of tabular RL methods has remained at a very low level. While many papers in the field report the effectiveness of utilizing DRL (e.g. [98], [97], [16], [52], ...), a notable success was a REINFORCE RS achieving good results in an online study [15], leading to YouTube[7] employing the system for video recommendation. According to the authors, this measure has led to the "largest single launch improvement [...] for two years" [1]. However, due to the news domain being underrepresented in the RSs research field overall, publications on Deep Reinforcement Learning News Recommender System (DRLNRS) are exceedingly rare ([99] and [50]).

The recommendation problem can be formulated as an MDP, and embedded into the RL framework laid out in Chapter 2, as follows:

- *Agent*: The agent assumes the role of the RS. When a user requests recommendations (e.g. by logging into the system, or explicitly entering the "For You"-page of the system, etc.), the agent determines which items to suggest, based on the user's profile. Usually, it has to fetch this profile from some internal memory.
- *Environment*: The environment is comprised of the sets of users and items of the RS.
- *State*: The state is the aforementioned user profile, i.e. a representation of the user that is requesting recommendations, along with contextual information. As already discussed in the presentation of SOTA NRSs in section 5.2, the user encoding typically consists of an embedding of previously transacted items, but can also

[7] https://www.youtube.com/

include features of the user itself, such as demographic information, or self-ascribed interests. Contextual information can be e.g. a timestamp, or a location. Clearly, this puts DRLRSs into the category of content-based methods, or hybrid methods, due to the use of function approximation, as discussed in Section 5.2.

- *Action*: Serving a user with a recommendation constitutes an action. The recommendation can consist of a single item, or a slate of items. The concrete action space depends on the specific domain and application. It can be discrete, e.g. consisting of a set of candidate items $\mathcal{C}$, and recommending each item is an action, i.e. $|\mathcal{A}| = \mathcal{C}$. But it could also be continuous, e.g. when the agent produces an n-dimensional vector that represents the ideal item, and subsequently uses it to recommend the closest item(s) from a set of candidates.

- *Reward*: The reward returned by the environment depends on the user's feedback to the recommended item, which can be either explicit or implicit, as already discussed. The reward design can also include contextual information, e.g. how long a user actively participates in the system. Note that the reward drives the training of the RS, with the evaluation metrics discussed in Section 3.3 having no impact on its development (unless they are embedded into the reward).

- *Environment Dynamics*: The environment dynamics, encapsulated in the four-argument function p (Equation 2.1), are the (generally) unknown probabilities of a user interacting with a recommended item, and rating it a certain way. These dynamics depend on the user's profile and the specific item that was recommended (i.e. state and action).

- *Model*: There are few works on model-based DRLRSs. Accurately determining the aforementioned environment dynamics is difficult in practical RSs, and using historical transaction data to estimate them further increases the computational complexity and cost of the system. Furthermore, while the agent adapts to dynamic changes in the environment, a model would require periodic re-training.

- *Task*: An agent's task begins with a user requesting a recommendation, and ends with the user exiting the system, or otherwise indicating that they no longer want to receive recommendations for now. One can assume that the user will exit the system in a timely manner and not request recommendations indefinitely. Another possibility is that the interaction ends, when all available items have been recommended or ranked. Therefore, tasks are episodic.

- *Discount Factor*: The discount factor γ is paramount, because it must encapsulate the evaluation metrics used for recommendations of multiple items in a ranked list. Concretely, when an agent sequentially recommends items to a user, the discount factor ensures that the agent's goal is to recommend interesting items first, and less interesting items later. When a recommended item draws a high reward from a user, e.g. in the form of a high rating, it should be dampened, when the item could have been recommended earlier. In contrast, when a user is dissatisfied with

a recommendation, it should matter less when it is one of the last items out of a finite set of candidates the agent has chosen to recommend. A short-sighted agent ($\gamma = 0$) is undesirable, since it will only care about the very first item it recommends. An undiscounted return ($\gamma = 1$) does not work in an RS, because the agent will have no notion of recommendation order. Given a set of recommended items, the agent would receive the same return for any sequence of suggestions. Instead, discounting will teach it to recommend good items first.

Aside from the obvious choice of which DRL algorithm to use (see Chapter 4), there are four key components to a DRLRS that must be adapted to the domain and available data. Firstly, the state representation design. As discussed in chapter 2, state is a loose concept in RL, with the only requirement being the Markov property. Of course, how to construct a user profile is a key question in regular RSs as well, as we have discussed in Section 5.2. Secondly, the action space design. As mentioned above, there are usually several ways of deciding what designates an action and the action space. Thirdly, the reward function design, which is an issue all RL applications face. And lastly, the environment design. Depending on the available data, there are several possibilities for how to train and evaluate a DRLRS agent. In addition, just like with regular RSs, conducting an online study is possible, but rarely done, due to the already mentioned restrictions. Since the design of these three components is highly domain- and data-dependent, we will solely discuss our own approaches in the practical part of this thesis, in Chapter 7.

CHAPTER 6

Data: Microsoft News Dataset

In this chapter, we will first briefly present and discuss the principal dataset used in this thesis, as well as summarize the accompanying paper. Subsequently, we will outline the approaches and results of our own preprocessing and data exploration, along with various visualizations.

6.1 Introduction to MIND and MIND Paper

The principal dataset used in this work is the **MI**crosoft **N**ews **D**ataset, "MIND" for short [93]. It is available at the MIND homepage `https://msnews.github.io/` for download, or via the Microsoft Azure Open Datasets cloud platform[1]. MIND was released in 2020 along with an accompanying paper, by researchers from Microsoft Research Asia, Microsoft and Tsinghua University. In the paper, the authors present the dataset, compare it to similar existing datasets and outline their shortcomings, as well as discuss various approaches to the news recommendation problem. They also provide experimental results for some of the methods presented in Section 5.2. As already mentioned, some of the authors have also been involved in the development of those methods. Furthermore, a competition was launched in 2020, encouraging researchers to use the MIND dataset as a testbed for NRSs. Through this competition, hosted via Codalab[2], predictions on the test set, for which the true labels cannot be downloaded, are automatically scored. Along with this competition went a call for papers and technical reports to be submitted for a workshop that was held on April 14th, 2021. The research group behind the dataset dubbed this the "1st International Workshop on News Recommendation and Intelligence"[3], albeit the 10th installment of the International Workshop on News Recommendation and Analytics,

[1] `https://learn.microsoft.com/en-us/azure/open-datasets/dataset-microsoft-news?tabs=azureml-opendatasets`

[2] `https://codalab.lisn.upsaclay.fr/competitions/420`

[3] `https://msnews.github.io/workshop.html`

"INRA" for short, took place in 2022[4] [63]. While the official competition is over, the submission process can still be used to obtain scores for predictions on the test set, which can be submitted to a public leaderboard[5].

According to the authors, the MIND dataset was built by collecting user behavior logs of the Microsoft News website[6]. It is supposed to contain impression data from 1 million randomly sampled users. The authors claim that only users, who had at least 5 news clicks within the time period October 12th to November 22nd 2019 (6 weeks, from a Saturday 00:00 am to a Friday 11:59 pm), were taken into account. We will refer to the aforementioned 6-week period as the "data period".

The dataset is already split into a training-, a validation/development- and a test-set. The impressions occurring in the last week of the data period comprise the test set. Those occurring in the penultimate week, i.e. the fifth week, comprise the training set. The validation set was built by further splitting the training set. Impressions occurring on the last day of the fifth week make up the validation set, all others the training set. We will thus refer to the concatenation of the training and validation sets, i.e. the impression data from the complete fifth week, as the "full training set".

A single impression begins every time a user visits the Microsoft News homepage. It provides information on when the impression began, which news articles where shown to the user, as well as whether the user did, or did not, click on each individual item. While the authors do not specify this, we assume that an impression ends when the user exits the Microsoft News homepage. In order to allow the inference of user's interests, the data includes for each user a list of all news articles that the user clicked on in the past. The impressions that occurred in the first four weeks of the data period were used to construct these user histories in the full training set. The histories remain fixed within the full training set, meaning clicks are not immediately incorporated into a user's history. However, the clicks that occurred in the full training set are added to the user histories in the test set, i.e. the histories in the test set were constructed from the impressions in the first five weeks of the data period. Obviously, the histories remain fixed within the test set as well, because the impressions in the test set are not labeled. Otherwise, we could simply read out, which articles a user clicked on or ignored during an impression. Note that while the authors did not do that for the full training set, continually updated user histories can be easily inferred from the data, by sorting by time and incrementally adding the news articles that a particular user clicked on to that user's previous history.

In order to summarize the general structure and construction process of the MIND dataset, as well as facilitate its understanding, we provide a visualization in Figure 6.1. Weeks 1 to 4 of the data period were only used to construct user histories. No impressions from that period are contained in the dataset. Importantly, histories only consist of clicks and hold no information as to which news were ignored by a user, even though this would

[4] https://research.idi.ntnu.no/NewsTech/INRA/
[5] https://msnews.github.io/#leaderboard
[6] https://microsoftnews.msn.com/

further shape a user's interest model. Week 5 is the full training set, with the last day comprising the validation subset. Week 6 is the test set, where user histories incorporate the clicks from Week 5 into the previous user histories, which is somewhat redundant.

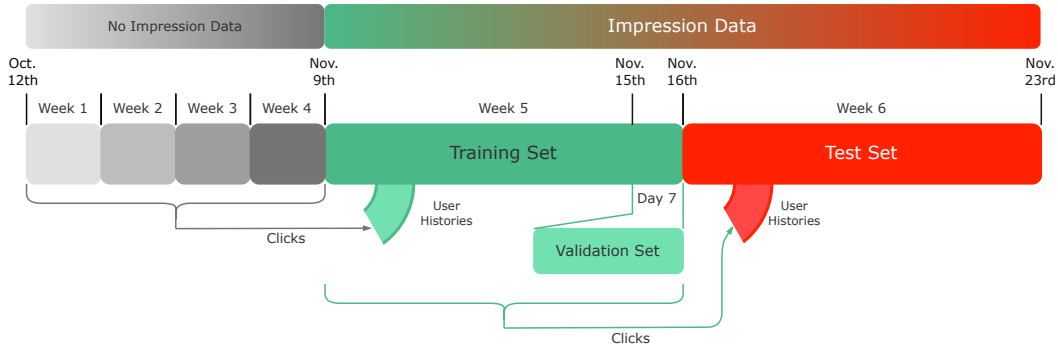


Figure 6.1: The general structure of the MIND dataset.

We will now explain the format of the impression data, by looking at concrete samples from the unprocessed dataset. In Table 6.1 are three impression samples, also referred to as behavior samples, from the validation set. Each entry consists of the following data points.

- `id`: A unique impression ID.
- `user_id`: A unique user ID.
- `time`: The timestamp at which the behavior occurred.
- `history`: A list of news articles (news IDs separated by whitespaces) that the user clicked on in the first four weeks of the data period. The list is sorted by click time, meaning the last read articles are at the end of the list. The authors only specify that the list is sorted, but we can infer from the updated histories in the test data that the sorting order is indeed ascending. This is important when it comes to building representations of users that aim to capture the sequential order of previously read news, which we will discuss in the subsequent chapters.
- `impression`: The list of news articles shown to the user, where a hyphen separates each news ID from its corresponding label, denoting whether the user clicked on (label `1`) or ignored (label `0`) the article. Unlike the reading histories, the news in an impression are shuffled. Note that the authors do not mention this in the accompanying paper, but on the corresponding GitHub page[7] linked on the MIND homepage.

[7] `https://github.com/msnews/msnews.github.io/blob/master/assets/doc/introduction.md`

We cut off both the `history` and the `impression` lists for presentation purposes. As discussed earlier, the entry in the `history` column is the same in each sample for a given user.

id	user_id	time	history	impression
46869	U1	11/15/2019 10:36:07 AM	N14639 N27258 N63237 ...	N121138-0 N104644-1 N7728-0 N56565-0 ...
29644	U1	11/15/2019 3:10:18 PM	N14639 N27258 N63237 ...	N55066-0 N130076-0 N18258-0 N80105-0 ...
2783	U1	11/15/2019 8:13:43 AM	N14639 N27258 N63237 ...	N19162-0 N83491-0 N121138-0 N94999-0 ...

Table 6.1: The behavior samples for user `U1` from the validation set.

In Table 6.2, we show behavior samples for the same user in the test set. The only difference in the data representation is that the `impression` list is unlabeled. Furthermore, as discussed, the user's history corresponds to the history in the full training set, extended by the articles the user clicked on in Week 4.

id	user_id	time	history	impression
1023170	U1	11/17/2019 6:03:46 AM	N14639 N27258 N63237 N112729 ...	N23931 N117631 N19135 ...
583877	U1	11/21/2019 6:59:37 PM	N14639 N27258 N63237 N112729 ...	N34469 N47516 N41213 ...
1136326	U1	11/22/2019 5:13:08 PM	N14639 N27258 N63237 N112729 ...	N65918 N79427 N76418 ...

Table 6.2: The behavior samples for user `U1` from the test set.

In addition to the behavior data, the MIND dataset includes data about the news articles occurring in the impressions. The data is also split into training-, validation- and test set, such that they cover all news articles occurring in the impressions of the corresponding set. Of course, this introduces redundancy, with some news article data being present in two or all sets.

We will now look at a concrete sample of the news article data from the unprocessed dataset. Table 6.3 shows a single sample from the validation set. Each entry consists of the following data points:

- `news_id`: A unique news ID.
- `category`: The category and ...
- `sub_category`: ... sub category of the news article. According to the authors, these categories were manually tagged by editors. Who these editors were is not discussed in the paper.
- `title`: The title and ...
- `abstract`: ... abstract of the news article.
- `url`: A URL leading to the full content body of the news article. The authors cite licensing issues[8] as the reason for not making the body available for direct

[8] `https://github.com/msnews/msnews.github.io/blob/master/assets/doc/introduction.md`

news_id	N88753
category	lifestyle
sub_category	lifestyleroyals
title	The Brands Queen Elizabeth, Prince Charles, and...
abstract	Shop the notebooks, jackets, and more that the ...
url	https://assets.msn.com/labs/mind/AAGH0ET.html
title_entities	[{"Label": "Prince Philip, Duke of Edinburgh", ...
abstract_entities	[]

Table 6.3: The news data sample from the validation set.

download. Already in May 2020, the authors admit to some URLs being expired or otherwise not accessible[9]. We further discuss this issue in Section 6.3.

- `title_entities`: Rich entities from the title and ...
- `abstract_entities`: ... abstract. How these entities were extracted is not exactly described by the authors. Instead, they only refer to an "internal NER and entity linking tool". We do not utilize these entities in the practical part of this thesis. There are many missing values, and none of the presented SOTA approaches use that data.

In regards to the dataset's dimension, the authors state that the dataset contains 161,013 news articles, and 4,893,502 behavior samples (split into 2,186,683 samples in the training set, 365,200 samples in the validation set and 2,341,619 samples in the test set), including a total of 24,155,470 clicks. As already mentioned at the beginning of this section, the authors claim that the number of users amounts to 1 million. Furthermore, the paper offers some extracted statistics. We will discuss both in the next section.

6.2 Preprocessing and Exploration

6.2.1 Dimension

The version of the dataset we (last) downloaded on 09/05/2023 contains 130,379 news articles, and 4,979,946 behavior samples (split into 2,232,748 samples in the training set, 376,471 samples in the validation set and 2,370,727 samples in the test set). The number of clicked articles from impressions in the full training set amounts to 3,958,501. The sum of the lengths of the histories of all unique users in the full training set totals 13,742,917. This number corresponds to the amount of clicks that occurred in the first four weeks of the data period. Together, this aggregates to 17,701,418 clicks, with approx. 22% of these being part of an impression.

[9] See commit: https://github.com/msnews/msnews.github.io/commit/49dd36e7727c16f0122d94e606c1a3a1bc20e5a2

Clearly, there are discrepancies between our findings and the numbers reported by the authors in the MIND paper. Firstly, there are more than 30k fewer news articles in our dataset. Secondly, the number of behavior samples is more than 86k higher than reported. Nonetheless, the number of clicks is smaller than reported by more than 6MM. Thirdly, the number of unique users amounts to 876,956, as opposed to 1MM. The root of these differences is unclear. There is no dataset versioning mechanism or documentation in place, so we cannot determine whether the reported statistics stem from the same dataset version as ours. However, an issue regarding the differing numbers of behavior samples was opened on GitHub[10] in November of 2020, but remains open as of now. Interestingly, many of the papers presented in Section 5.2 regurgitate the same numbers as the MIND paper, which begs the question whether their version and our version of the dataset are the same.

As already mentioned in the previous section, the news data is also split into redundant sets. We examined whether:

(a) All news IDs present in the behavior data are covered by a corresponding entry in the news data?

(b) Every news data sample occurs in the behavior data at least once?

In order to answer these questions, we constructed two sets. Firstly, the set of all news IDs, $allN$, is the union over unique news IDs in the news data splits. Secondly, the set of relevant news IDs, $relN$, is the union over unique news IDs occurring in the behavior data, either in an impression, a user's history or both. We found that 58% of IDs $\in relN$ are solely part of reading histories, 25% are solely part of impressions, and the remaining 17% are part of both. Since the difference $allN \setminus relN$ is the empty set, we conclude that (a) is true. Furthermore, since $allN = relN$, we conclude that (b) is also true. These conclusion are important, because they ensure that we indeed have behavior data for each news data sample, and vice versa.

6.2.2 Preprocessing Steps

We provide a commented Jupyter Notebook that contains our entire preprocessing code, facilitating the reproduction of our preprocessing steps. It can be found in the corresponding GitHub repository[11].

We begin with the preprocessing of the behavior data. We differentiate in the code between necessary preprocessing, and the extraction of information for exploratory purposes. The necessary steps were: 1) converting the timestamp string into a datetime format, 2) splitting the history string into a list of news IDs, 3) splitting the impression into lists of clicked/ignored/shown news (for the test set, the distinction could not be

[10] https://github.com/msnews/msnews.github.io/issues/18

[11] https://github.com/d-vesely/drlnrs

made), 4) sorting the data by user ID and timestamp, and subsequently processing the user histories. In this last step, we extended each user history with the clicks that occurred in the user's previous impressions. As already mentioned, the authors did not do this, leaving all histories unchanged within a split. Especially since we aim to exploit the sequential nature of the news recommendation problem, we think this is an important step that should have been done by default. For exploratory purposes, we extracted a multitude of values for each impression, such as the lengths of user histories, the number of clicked and ignored news, etc. We also computed statistics for these values. The results are presented in the next subsection. Finally, we concatenated all three preprocessed data splits, keeping only the list of news shown to the user and the timestamp. We later used this concatenated data to obtain survival information for each news article, i.e. how many hours were between the first and last occurrence of an article.

For the news data, we first concatenate the three redundant splits into a single dataset. Again, we differentiate between necessary and exploratory preprocessing steps. The former were: 1) replacing missing abstract values with empty strings, 2) remapping the categories and subcategories. A few categories were mislabeled, and most subcategories were conjunctions of multiple words, which would be difficult for the embedding model to handle, so we replaced them with natural language keywords. For the embedding step, we also created additional columns, such as the concatenation of title and abstract. As we will discuss in the next chapter, we utilized a pre-trained large language model for the embedding of the textual data as a whole. Therefore, tokenization, lower-casing, stopword-removal and other typical preprocessing steps done in NLP projects, were not necessary. Nonetheless, we included them in the exploratory preprocessing, in order to obtain values such as the lengths of titles and abstracts, but also to facilitate the use of word embeddings in the future. Finally, we utilized the aforementioned concatenated data, as well as the full training data, to obtain survival data and engagement data, i.e. the total number of clicks, appearances, etc., for each news item.

Aside from the irrelevant entity data columns, and the already mentioned missing abstracts, we found that user histories were missing in a total of 57,335 impression (approx. 2.2% of all impressions), for 14,086 unique users. It is unclear whether these missing values designate users that have not yet read any articles, or whether it constitutes an error. However, the fact that 17.6% of all initial user histories contain between 1 and 4 news items directly contradicts the authors' claim that a minimum of 5 clicks was required for inclusion in the dataset.

6.2.3 Exploration

Following the authors, we will present some interesting statistics extracted from the dataset in this subsection. Again, we provide[12] two separate notebooks, one for the behavior data and one for the news data, that can be used to reproduce our visualizations.

[12] https://github.com/d-vesely/drlnrs

Again, we begin with the behavior data. Figures 6.2(a) and 6.2(b) show histograms for the length of user histories and the number of impressions per user. 75% of users have histories of length 22 or less, and 90% of users stay at or below 42, but there are some users with very long histories as well, the maximum being more than 800. 1 is the most frequently occurring amount of impressions per user. Most impression are rather short, as can be seen in Figure 6.2(c). Figure 6.2(d) shows that in the vast majority of cases, users click only on a single news item per impression. Consequently, the percentage of ignored news is typically extremely high, and also increases the longer the impression is, as can be seen in Figures 6.2(e) and 6.2(f) respectively. Last but not least, Figures 6.2(g) and 6.2(h) show that most impression occur in the morning (5-11 am) and during the day (11am-5pm), only few in the evening (5-11pm) and even less at night (11pm-5am).

We continue with the news data. Figures 6.3(a), 6.3(b) and 6.3(c) show histograms of the lengths (as the number of words) of title, abstract, and the concatenation of both, respectively. It also shows how the length is affected when stopwords are removed. The concatenation of title and length consists of 95 or fewer characters for 99% of all articles. So during the embedding process (see Section 7.1.1), the textual data should rarely be cut off, despite language models usually introducing more tokens than just words. These graphics correspond to those shown in the MIND paper [93], with the exception of the spike for abstracts that have 0 length, i.e. a missing abstract. We assume that the authors omitted those values. A histogram of the survival times are shown in Figure 6.3(d). 90% of articles have a survival time below 86 hours, 75% below 32 hours, and 50% below 17 hours. Both these stats and the visualization only include articles that allow for survival time measurement, i.e. occur in at least two impressions. Finally, the Figures 6.3(e), 6.3(f) and 6.3(g) show histograms of the number of times articles are shown and clicked, as well as the engagement percentage. Again, we omit articles that only occur in user histories and not in an impression. 50% of articles are shown 54 or fewer times. 50% of articles rack up 5 or fewer clicks. However, some articles are shown and/or clicked thousands of times. Approximately 10% of articles are shown more than 4000 times. Approximately 1% of articles are clicked more than 4000 times. Approximately 50% of articles are clicked less than 4% of times that they are shown, and 99% of articles are clicked $\leq \frac{1}{3}$ times that they are shown.

6.2.4 Numerical News Features

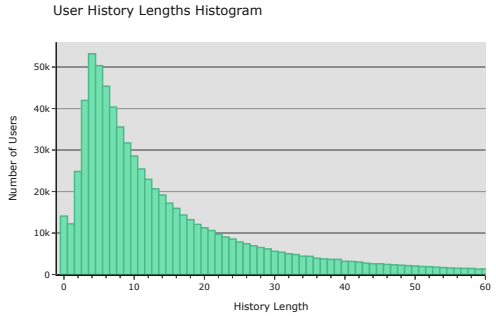
As a byproduct of the previously discussed preprocessing and exploration steps, we extracted 11 numerical features for each news item. Firstly, the lengths of title and abstract, both individually and concatenated, with and without stopwords (6 features). Secondly, engagement numbers, concretely the number of times an article was clicked, ignored, and shown throughout the dataset's time period, as well as the percentage of showings that led to clicks, indicating its popularity or lack thereof (4 features). Lastly, the item's survival time. Note that the engagement features are collected from the entire training data split. Usually, an RL agent would be trained online and thus not have access to these numbers a priori. The same is the case for the final survival time, which

inherently will always be undetermined when an article is in the candidate set.

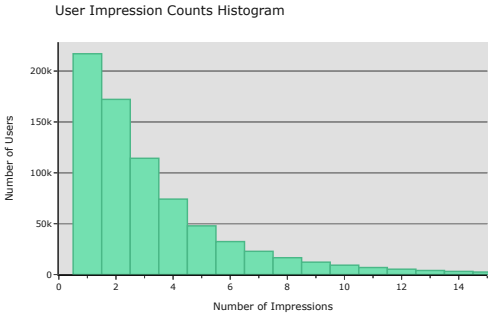
6.3 Discussion and Critique

We want to present a set of points about the MIND dataset that we find worth critiquing:

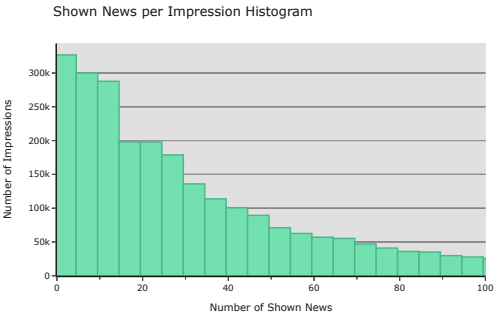
- First and foremost, the discrepancies between the authors' claims and our own findings regarding the contents of the dataset pose a reproducibility issue. Comparisons between our results and those presented in other works carry less value, as long as it is unclear, whether the utilized datasets are equivalent. If a different version of the dataset was used, we cannot account for it, due to the lack of a transparent versioning system.
- At the time of writing, all URLs seem to have expired (naturally, we have not tested all of them). This means that the news article bodies are not accessible. Furthermore, additional information cannot be extracted, such as the article's author, publisher, release date, etc., which presumably could have been scraped from the corresponding website. As already mentioned, the authors refer to licensing issues to justify the exclusion of article bodies from the dataset. Nevertheless, we want to emphasize the value of the aforementioned information. For instance, publishers and authors could be used for beyond accuracy evaluation and training, to ensure readers do not consume news content from a single source.
- We do not understand the authors' decision to shuffle the order of news articles in the impressions. Especially for the application of RL, we deem the sequentiality of the impressions to be valuable information. For instance, a user might be interested in a current event and read an article about it. Subsequently, the user might choose to skip other news about said event, since they already feel adequately informed. Without being shown the order of appearance, a recommender agent will not be able to learn about these cases. Instead, it could face impressions, where a user seems to skip an article that would otherwise be interesting, only to later read a similar article, and vice versa. Of course, this is only an issue if our assumption about this user behavior is correct.
- The authors provide no information on the selection process for the shown articles. It would be interesting to know, whether these were randomly selected, or recommended. And if so, whether the candidates were selected all at once and ranked, or adapted during the impression.
- Similarly, insufficient documentation is provided regarding the information provided to the user before clicking on an article, such as whether the abstract is visible or just the article's title, whether they are presented in a list or a grid, whether some articles receive more screen space (e.g. breaking news), and so on.



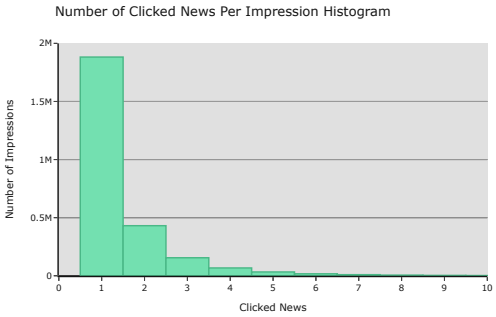
(a) User history lengths.



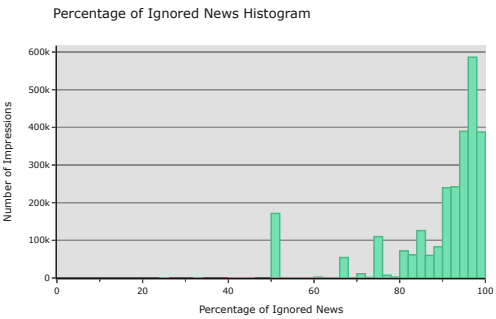
(b) User impression counts.



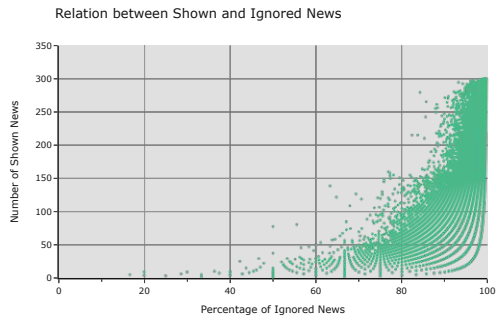
(c) Shown news per impression.



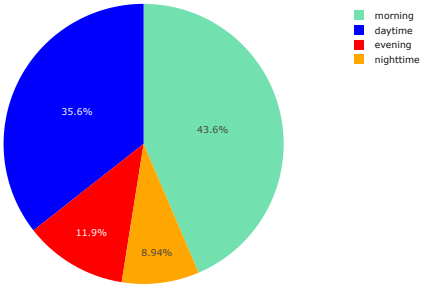
(d) Clicked news per impression.



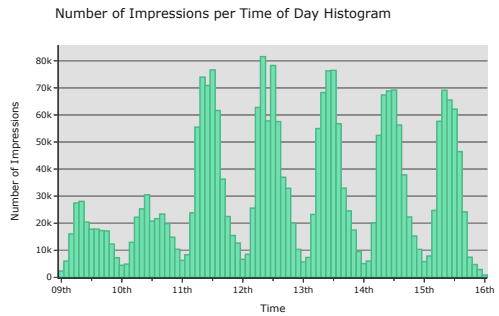
(e) Percentage of ignored news.



(f) Relation between shown and ignored news.

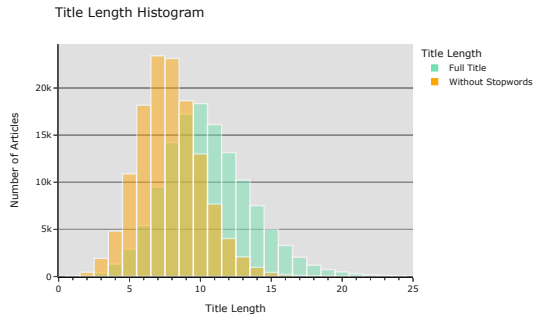


(g) Percentage of impressions per time category.

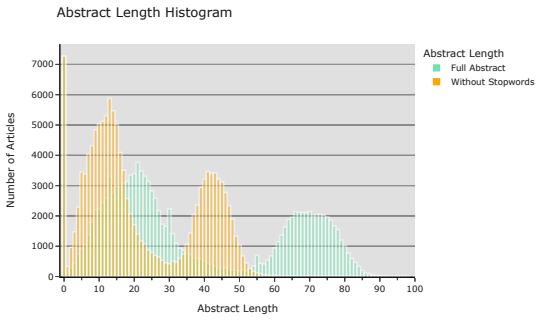


(h) Number of impressions per time of day.

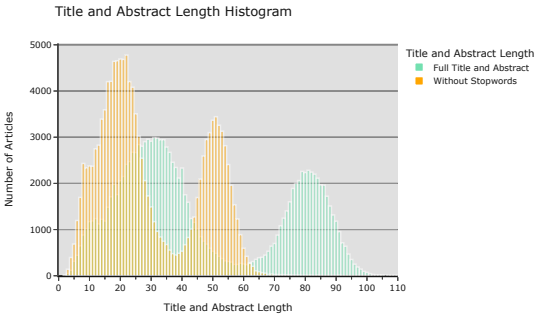
Figure 6.2: Visualizations of exploratory behavior data.



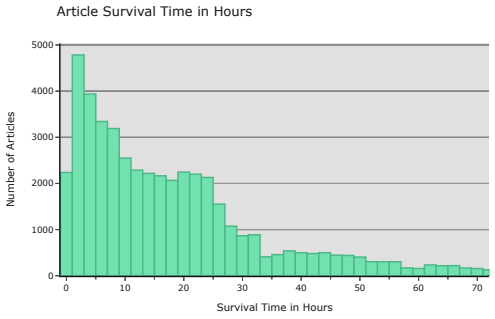
(a) Title lengths.



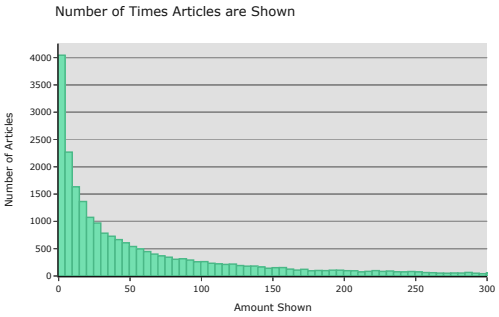
(b) Abstract lengths.



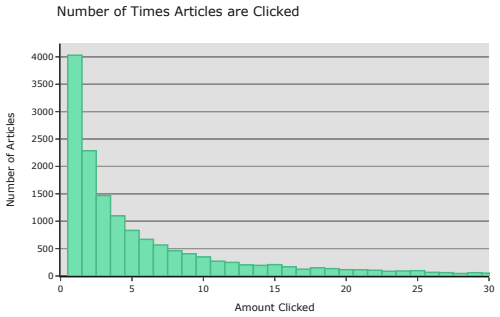
(c) Lengths of concatenated title and abstract.



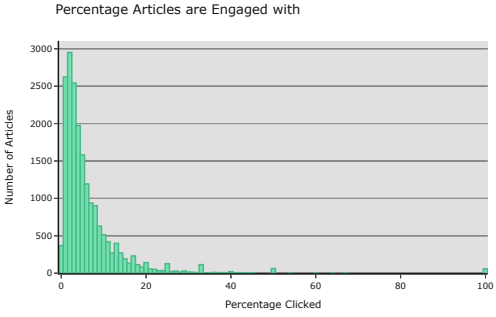
(d) Article survival times.



(e) Show-numbers for articles.



(f) Click-numbers for article.



(g) Engagement percentages.

Figure 6.3: Visualizations of exploratory news data.

CHAPTER 7

DRLNRS: Comparative Analysis

7.1 RL Framework

A key aspect of the practical part of this thesis was the construction of an RL framework around the MIND dataset. The main goals were readability, extensibility and reproducibility. Our code provides a variety of news and user encoders, that can be easily modified, or extended by new approaches. We provide a set of implemented DRL algorithms to be applied on the MIND dataset. We do not rely on third-party RL libraries, thus ensuring readability. Jupyter notebooks facilitate the training and evaluation of agents, as well as the production of result visualizations. All of the code written and used in the context of this thesis is open-source and available via GitHub[1]. In this section, we will outline and discuss the individual components that make up the RL framework.

7.1.1 News Encoder

In general, the news encoder uses the available data for a news article, as well as features extracted from the behavior logs, which we have outlined in Section 6.2.4, to produce an embedding vector. We have tested a total amount of 9 approaches, made up of different combinations of input data. For the embedding of textual input, we utilized pre-trained language models available via the `sentence-transformers` library[2] [69]. For the embedding of title and/or abstract, we opted for the model `all-mpnet-base-v2` with the maximum number of tokens set to 256, which produces 768-dimensional encodings. Given the length distribution of title and abstract, it is safe to assume that the vast majority of inputs will not require any cutoff. For the embedding of category, or subcategory, we opted for the model `all-MiniLM-L12-v2`, which produces 384-dimensional encodings. Furthermore, we have also tested OpenAI's embedding model `text-embedding-ada-002`

[1] `https://github.com/d-vesely/drlnrs`
[2] `https://www.sbert.net/docs/pretrained_models.html`

[31], which generates 1536-dimensional vectors, with a maximum number of input tokens of 8191[3]. We have done this to approximate the impact of the used PLM on performance, following [88]. Note that obtaining these embeddings requires a (very small) payment for the use of the OpenAI API. The following enumeration summarizes all tested news encoding schemes, with each one visualized in Figure 7.1.

1. Embedding of the title.
2. Embedding of the abstract.
3. Concatenation of the embeddings of title and abstract.
4. Embedding of the concatenation of title and abstract.
5. Embedding of the category, concatenated with the embedding of title and abstract.
6. One-hot encoding of the category, concatenated with the embedding of title and abstract.
7. Embedding of the subcategory, concatenated with the one-hot encoding of the category and the embedding of title and abstract.
8. Embedding of the concatenation of category, subcategory, title and abstract, in that order. The concatenation is a formatted string, of the following form:

formatted_string.py

```
f"category: {news['category']}, subcategory: {news['sub_category']}. \
    title: {news['title']}, abstract: {news['abstract']}"
```

Algorithm 7.1: The formatted string used for the concatenation of all textual information of an article.

9. Constructed features, concatenated with the embedding of title and abstract. For the reasons discussed in Section 6.2.4, we do not use an item's survival time. However, we test both including and omitting the engagement features. Each feature was standardized to have a mean of 0 and a standard deviation of 1.

[3] https://platform.openai.com/docs/guides/embeddings/what-are-embeddings

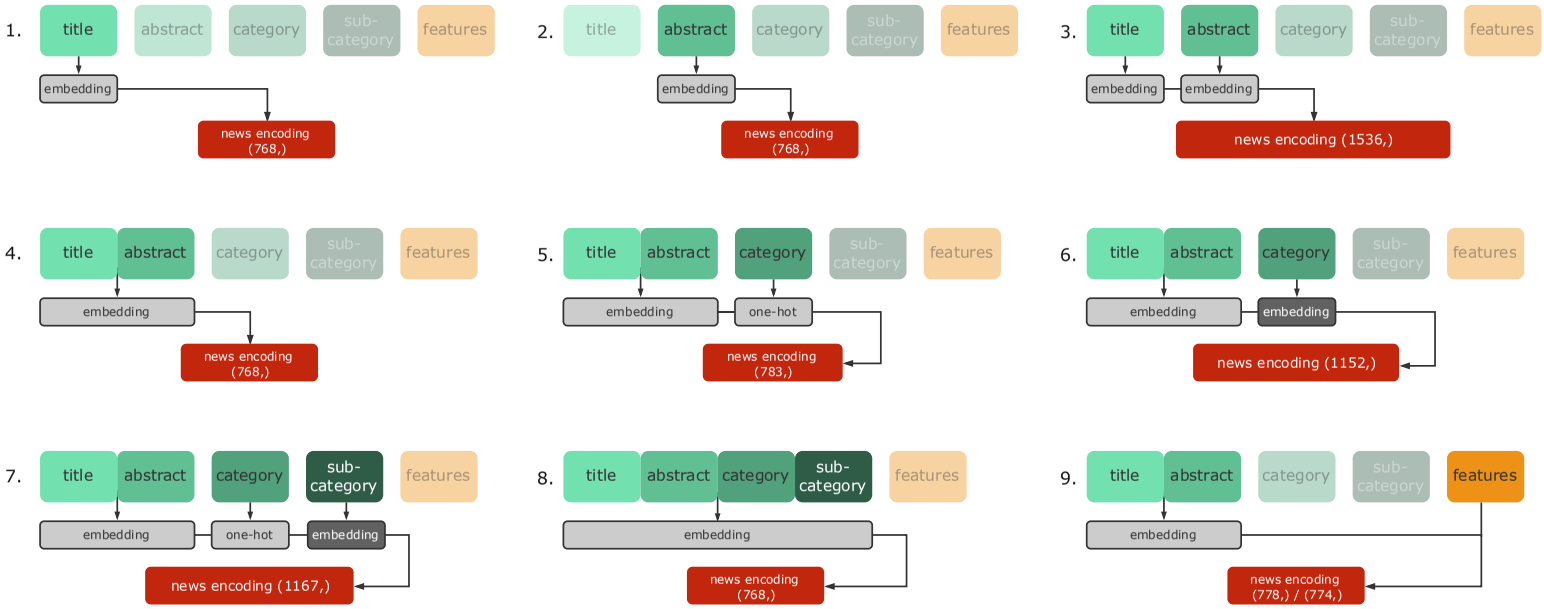


Figure 7.1: News encoder variants.

7.1.2 User Encoder

Given the data available to us in the MIND dataset, our state is solely based on the ordered list of previously read articles, of the user that is currently requesting recommendations. We have tested a total amount of 8 approaches to encoding the list. Besides testing which method yields the best performance, we also aimed to get insights into the degree to which news recommendation is a sequential problem. Our selection of approaches was inspired by the SOTA methods presented in Section 5.2, as well as by [53]. Again, the following enumeration summarizes all tested user encoding schemes, with each one visualized in Figure 7.2.

1. The (weighted) mean of all news encoding vectors. Each article is weighted with a value α^k, where k denotes the article's position in the reading history. The most recently read item has rank $k = 1$.
2. Multi-head self-attention with 8 attention heads, followed by a mean-pooling over all attention head outputs.
3. Multi-head self-attention with 8 attention heads, followed by additive attention pooling.
4. The concatenation of three quantiles over all news encoding vectors (0.25, 0.5 and 0.75).
5. The final last hidden state of a bi-directional GRU network.
6. The mean over all last hidden states of a bi-directional GRU network.
7. Additive attention applied to all final last hidden states of a bi-directional GRU network.
8. The concatenation of the encoding of the last read article, the weighted mean over the encodings of the last 5 read articles (short-term) and the weighted mean over all other articles, besides the last 5 (long-term). We call this encoding scheme "LTSTL", for long-term-short-term-last.

When possible, i.e. for encoding schemes 1., 4. and 8., we processed the user's entire reading history. Otherwise, we limited the history to the last 40 read articles. Without this limit, mini-batch processing is not possible. Histories shorter than 40 articles were padded.

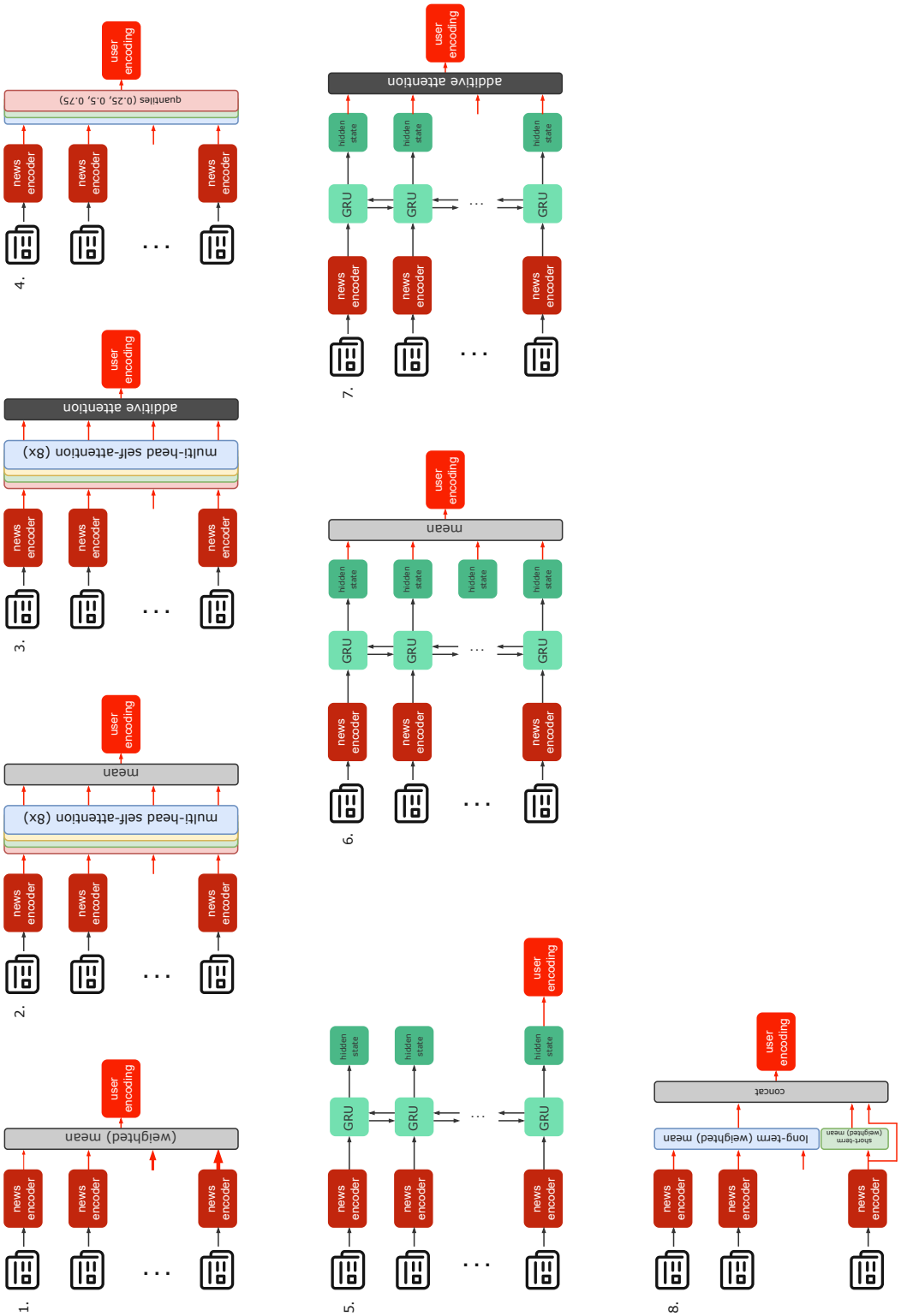


Figure 7.2: User encoder variants.

7.1.3 Action Space

While there are several ways of encoding the state, it is straightforward to decide, which information is used to designate the state. In contrast, we find there are multiple approaches to designing the action space, both discrete and continuous, with the given MIND dataset. First of all, it is important to re-emphasize the points made in Section 6.3. The impression logs are shuffled, which means we cannot reconstruct the order of consumption. We also have no information on whether the recommended items were all selected as candidates immediately when the user sent a request for recommendation, or if the candidate set was incrementally built or adapted.

In order to obtain a discrete action space, one possibility would be to assume a fixed amount of candidates c, from which the most suitable item is recommended. The resulting empty slot would then be filled with a new candidate item. However, we deem this approach to be too restrictive with a high potential of losing valuable information, especially in light of the fact that the order of consumption is unknown. Furthermore, even if the order was known, it is unclear whether the MIND dataset was collected with an RS in place, or whether the users were simply served newly released articles. Last but not least, if c is set too small, it can be automatically impossible to recommend the best item first, if it is not in the initial set of candidates. Therefore, we determined that the safest approach would be to assume that all items were candidates a priori. Unfortunately, this also means that the number of items available for recommendation, i.e. the number of actions, constantly changes and can be very high (see Figure 6.2(c), showing the distribution of impression lengths). Nonetheless, this approach constitutes a large, and potentially unbounded, discrete action space. Yet another design decision to be made, is how the RL agent is informed of the available actions. Concretely, how the news encodings of each candidate item are input into the LDNN function approximator that is inherently a central component in each DRL agent (see Chapter 4). The authors of [38] suggest to pre-allocate slots in the LDNN input for the maximum amount of available actions. While this is a feasible approach in their work, where the number of actions also varies per state, but is at most 4, we deem it highly impractical for the large amounts of possible actions in our case. Using a sequence model for the candidates could be a way to mitigate the need for pre-allocation, but we expected the model to have trouble remembering and distinguishing the available items for large amounts. Therefore, we found the approach of individually processing all candidates as the only sensible one, albeit it is less efficient. Almost all of the tested algorithms that require a discrete action space just output a single action-value or quantile for a given candidate, and subsequently compare these values for all candidates, after processing each one. However, for the REINFORCE algorithm, we have tested the approach of emitting two values for each candidate, namely the probabilities for the two discrete actions "recommend" and "skip" that the agent faces for each candidate.

We have identified two options of defining a continuous action space. Firstly, following [51], we can define an action as a continuous vector of the same dimension as the vectors produced by the chosen news encoding. The ranking score for each candidate is then

the dot product between the action and the candidate's news encoding. Secondly, the authors of [25] also use a continuous vector of equivalent dimension as an encoded item, but propose an agent that learns to generate actions that serve as prototypical items. A prototype is then used to find a set of k candidates that resemble it most closely. This fixed, small set of candidates can then easily be regarded as a discrete action space. Furthermore, in the case of $k = 1$, the appropriate candidate can be selected immediately. For both cases, a design decision is whether to process the candidates as input, and if so, how. The aforementioned problems arise again, i.e. pre-allocation is usually infeasible and sequence models might have trouble handling long sequences of candidates. However, if each candidate is again processed separately, there is no added benefit to opting for a continuous action space over the discrete definition discussed before. Nevertheless, completely omitting the candidates in the input and only utilizing the state has value as well, because it can be utilized in a component of the RS that constructs the set of candidates. Especially in the news domain, distilling the large amount of available news articles into a small set of items that are potentially interesting to the user is a fundamental part of the recommendation process.

Note that, on a higher level, both cases ultimately lead to an action being the process of recommending an item to the requesting user. The approaches only differ in the way they select the item to recommend.

7.1.4 Reward Function

As already mentioned in the previous chapter, the reward should incorporate the feedback users have given in response to a recommended item. Since the MIND dataset does not provide any additional contextual information, such as user activeness, our reward function is purely based on the implicit feedback in the form of binary click values. As we have presented in Chapter 4, the reward is a key part of the update target. In cases of a reward of 0, no update occurs, in the sense that one estimate is assigned the value of another estimate. Of course, there are applications that solely assign a non-zero reward at the end of an episode, thus updating only a single value per episode. A typical example for such an application is an agent learning to play a board game. The actions up until a terminal state should be neither rewarded nor penalized, and a constant reward at every timestep would lead the agent to trying to reach a terminal state as slowly/quickly (depending on the sign of the reward) as possible. Instead, the only thing that should matter to that agent is whether the game was won or lost. In summary, the reward functions needs to be designed in a way that describes to the agent what to do, not how to do it. In the board game example, winning is the goal and it is up to the agent to learn which moves are best to reach it. However, in our case, simply using the binary click values as a reward would yield an agent that solely learns from successful recommendations. This would immensely prolong learning, especially due to the rarity of positive examples. Furthermore, unlike with board-games, our agent does not have "intermediary" actions that it can take in pursuit of its goal. Recommending interesting items is its goal, and each action constitutes an attempt at reaching it. Therefore, we

will return a reward of -1, if the recommended item was not clicked, and +1 otherwise.

7.1.5 Environment and Training Process

Since we are utilizing a dataset of historical transactions to train an RL agent, one can argue that we are operating in a blend of SL and RL. Furthermore, in order to fit into the framework of the MIND benchmark, we intend to train the agent with data from the training split, and evaluate it on data from the development, and test splits. This is also fairly atypical for RL agents, which are usually continuously evaluated during training, by monitoring their obtained return. We determined two approaches to using the MIND dataset to train an RL agent:

1. *Simulation*: Instead of directly interacting with the environment, the agent would be placed in a simulator. Following the order of occurrence, the agent would receive simulated requests for recommendation by individual users. As already discussed, the initial state would be the requesting user's reading history. As discussed before, the entire set of news items that the user has interacted with during the impression would serve as the set of candidate items that are up for recommendation. The agent would then recommend items one by one, until eventually exhausting all candidate items, at which point the episode terminates and a new recommendation request is issued. It would receive rewards for each recommended item from the simulator, depending on the ground truth recorded in the dataset. This approach closely resembles the regular RL training process, allowing the application of exploration techniques and prioritized experience replay.

2. *Offline*: The second possibility is to randomly construct a replay memory from the dataset, and subsequently use it to train the agent. By randomly constructing a recommendation order for each impression, one can obtain a new dataset consisting of tuples of the form (state, action, reward, next-state, candidates), i.e. the user's history, the recommended news item, whether the user clicked on the recommendation, the subsequent new user's history and the remaining set of candidates. For example, the Tables 7.1 and 7.2 show a fictional entry in the dataset, and the resulting entries in the constructed replay memory. Note that the set of seen news items is shuffled and then recommended one-by-one. The current history (state) can easily be reconstructed from the next history (next state) and the obtained reward. Furthermore, the reward is simply a copy of the binary clicked value, and can be adapted at a later stage (see previous Section 7.1.4). The resulting replay memory dataset is extremely large, since each row in the MIND dataset is expanded to n rows, where n is the number of items in the impression (see Figure 6.2(c) for the histogram of the amount of shown news per impression). However, this approach allows us to oversample positive results, i.e. cases where a user clicked on the recommended item, which the MIND competition winners identify as an important factor for training in their technical report [18]. To that end, we split the replay memory into a positive and a negative memory, where the former is

comprised of recommendations that were clicked (resulting in 3,383,656 rows), and the latter includes those that were ignored. In essence, this approach constitutes a purely exploratory agent (random recommendations) collecting experience data, and using that data to train a more successful agent, by oversampling experiences with positive rewards. The disadvantage is that exploration techniques and prioritized experience replay cannot be used.

Table 7.1: Fictional entry in the MIND dataset, consisting of the requesting user's history and the impression log.

history	impression
N1 N2 N3	N4-0 N5-0 N6-1

Table 7.2: The resulting entries in the randomly constructed replay memory.

recommended	reward	next_history	next_candidates
N5	0	N1 N2 N3	N6 N4
N6	1	N1 N2 N3 N6	N4
N4	0	N1 N2 N3 N6	[ ]

We opted for the second approach, because we deemed it more efficient and assumed it would yield lower training times. Furthermore, it strongly facilitates oversampling, which we hypothesized to be paramount, and our results reflect that. It is important to emphasize that, ultimately, the agent has no impact on which of the recommended items the user will click on, since that is pre-determined in the dataset. It can only learn to recommend interesting items first. However, as we have noted in Chapter 6, the order of recommendation could have an effect on a user's click behavior. Naturally, only an online training approach with an online study could mitigate this issue.

We have also constructed an episodic replay memory, where each entry represents an entire episode, as opposed to a single environment transition step. This allows the usage of DRL algorithms that work with entire trajectories, such as REINFORCE.

7.2 DRL Algorithms

We have implemented a total of 8 DRL algorithms and applied them to the MIND news recommendation problem, namely DQN, C51, QR-DQN, IQN, FQF, DDPG, TD3 and REINFORCE. The necessary theoretical background for these algorithms was presented in Chapter 4. As already mentioned, we have not relied on third-party RL libraries. In our opinion, these libraries are too restrictive regarding the action- and state-spaces, and the implementation of the environment, making our offline approach difficult or impossible. Furthermore, we did not find a library that would offer all DRL algorithms that we have implemented. Therefore, our approach of implementing everything from scratch ensures comparability, readability and further facilitates reproducibility, due to the lack of a dependency on volatile RL libraries (see also the brief discussion in Section 8.3).

While the used architectures for each algorithm can be easily found in the code itself, the following Figure 7.3 visualizes the LDNN designs. We used a hidden size of 4096 for all experiments.

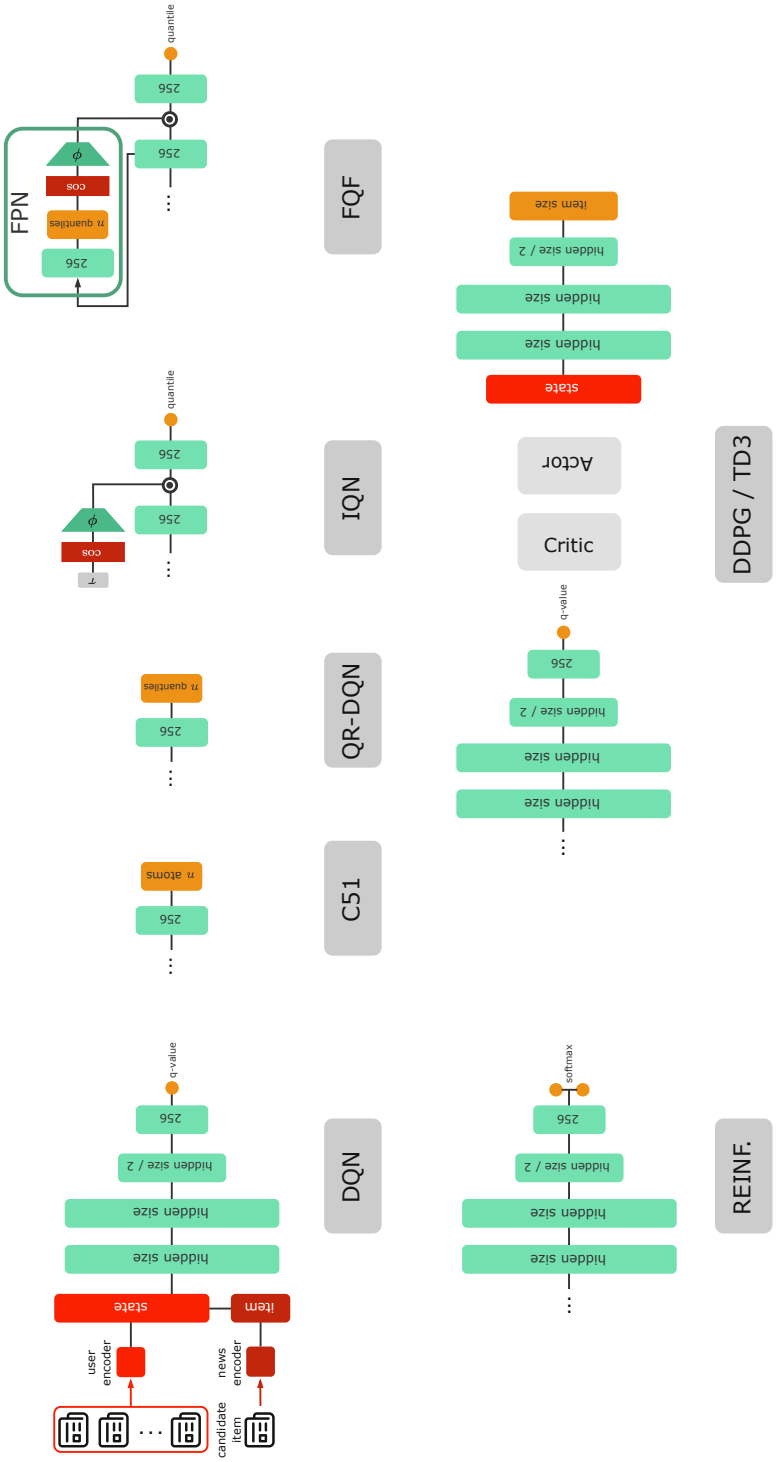


Figure 7.3: Overview over the LDNNs used for each DRL algorithm.

7.3 Experiments

We train all agents solely on replay memory that was constructed with the training data. As discussed earlier, the replay memory is separated into positive and negative memory, which we sample from independently. Due to the size of the entire negative memory, we used a sufficiently large subset during training. The exact subset can, of course, be reproduced. The development data is used for evaluation. For each impression, the evaluated agent is supplied with the initial user history, as well as the set of candidates occurring in the impression, which constitutes the set of available actions. Based on the agent's value estimates, the actions are ranked from best to worst, which constitutes an order of recommendation. We have also tested a sequential evaluation approach, where we assume that the user always clicks on the first 3 recommended items, and the user encoding is incrementally regenerated with each of these 3 news items incorporated into the user's reading history. Therefore, the agent's value estimates adapt during evaluation. The purpose was to further test the effect of recently read news on the user's choice. Intuitively, an agent would initially attach similar value to similar articles. However, once the user reads one article, the value of the other should drop significantly, since the user is presumably not interested in re-reading the same story. Of course, the assumption that the first 3 recommendations are accepted is strong. Unfortunately, we cannot rely on the real click labels. As outlined in Section 6.3, we don't know whether articles were skipped due to a lack of interest, or due to the recommendation order.

We train each agent for 6 million steps, saving model checkpoints on every millionth step. In addition, we save three early checkpoints after 10k, 100k and 200k training steps, in order to better evaluate, how quickly an algorithm progresses. Since the aim of this work is a comparative analysis of different DRL algorithms applied to a news recommendation problem, we refrain from using the evaluation metrics used in the MIND benchmark. Instead, we use the average obtained discounted return as the primary performance measure, which is common in RL experiments. For the evaluation, we always use a discount factor of $\gamma = 0.9$. As opposed to training, the discount value is unimportant during evaluation, as long as it is less than 1 (which would yield the same return, regardless of recommendation order) and consistent for all experiments.

We repeated each training twice with different random seeds and report the mean and standard deviation of the results. All experiments, including the construction of the replay memory, were conducted on a PC with a i7-13700K CPU and 64GB of RAM, as well as an NVIDIA GeForce RTX 3080 GPU, which was used for deep learning, using the PyTorch library [64]. For a complete list of Python packages and versions, refer to the GitHub repository[4].

[4] https://github.com/d-vesely/drlnrs

7.4 Results

First and foremost, we established a baseline result, obtained by generating two random recommendation orders and evaluating each one. The mean of the two average discounted returns was 0.5521, which forms our baseline. For visual reasons, we do not display this baseline in any of the result visualizations, since the obtained results are much higher, even after few training steps. The baseline can be reproduced in the provided evaluation notebook.

7.4.1 Hyperparameters

We started our experiments by varying the set of RL hyperparameters, while keeping the RL algorithm, the user encoder and the news encoder fixed. We used regular DQN, a weighted mean of news encodings, and the embedding of the concatenation of title and abstract, for the aforementioned components. Subsequently, the hyperparameter settings that we deemed best as a result of these experiments were reused in later experiments. Of course, the assumption that varying the user encoder, news encoder, or the DRL algorithm has no effect on which hyperparameter settings yield the best results is left unproven. Nonetheless, we believe that this approach is valid for the following reasons:

1. The main goal of this thesis is a comparative performance analysis of DRL algorithms, as well as of user encoders and news encoders. To that end, a set of hyperparameter values that remains consistent through all experiments is required. These values need not necessarily be those that yield the best overall results. Nonetheless, the argument can be made that the parameters were set using DQN, giving it an inherent advantage over other algorithms, which might demand other settings. We do not think this is a serious issue, because ...

2. On the one hand, most hyperparameters, as we will see, have little effect on the performance.

3. On the other hand, some hyperparameters have an effect that we deem strong enough to be universal for all algorithms.

We conducted neither grid search, nor random search. Instead, we opted for an initial hyperparameter setting, and gradually varied each hyperparameter to test its impact and obtain a well-performing value for it. This assumes a certain level of independence between hyperparameters, which we are willing to take on for the first of the above listed reasons. Note that we always picked the parameter value that led to the best performance, even if changes were marginal. Some results were so close that, given other seeds or additional runs, another value choice might have taken the lead.

The following enumeration discusses the results of each graphic in Figure 7.4, where the item label corresponds to the respective caption label:

(a) We began by testing two learning rates, i.e. $1e^{-4}$ and $1e^{-5}$. The former yields much better results, but already leads to steep improvements in the first checkpoints, while performance decreases after 3MM training steps. Therefore, we did not test an even smaller learning rate.

(b) The positive memory preference is the probability, with which the agent chooses to sample a batch of positive experiences, i.e. cases where the user clicked on a recommended item, as opposed to negative experiences. In other words, this parameter tunes oversampling. We tested a PMP of 10%, 30%, 50% and 70%. Looking at the percentage of ignored news in Figure 6.2, all of these constitute some degree of oversampling. As hypothesized earlier, especially in comparison to other hyperparameters, the PMP has a huge impact on performance, with 30% being the best out of the tested values.

(c) A discount value γ that is on the lower side led to better performances, but there was little difference between a gamma of 0.65 and 0.80.

(d) We always applied learning rate decay (for details, consult the code). A slowly decaying LR outperformed one that decayed quicker, albeit the results were very similar. Furthermore, the quicker decay seemed more stable, in the sense that the performance score kept improving until the training ended.

(e) The frequency with which the target network is updated via direct copy had a small impact on performance, with updates after every 5k steps performing best, compared to every 10k and every 500 steps.

(f) In comparison with a direct target network copy, a soft target update conducted after every training step with $\tau = 0.01$ (see Section 4.5) led to a slightly better final performance.

(g) We also tested an adaptive PMP, one increasing the PMP by 0.04 every 500k steps, another increasing it by 0.1 every 1MM steps. The idea was to simulate the agent getting better at recommending good items first, as training progresses. While both methods improved the training speed and the second method obtained the best performance score, a constant PMP of 30% appeared to be more stable and improved steadily. Performance drops significantly beyond 4MM training steps, when using adaptive PMP.

(h) Last but not least, we tested the impact of the batch size, by comparing sizes of 16, 32 and 64. Smaller batch sizes performed better, especially at the early checkpoints. However, the performance scores were slightly more volatile, whereas a batch size of 64 generally led to a steadily improving score. Ultimately, due to the final measurements being quite similar, and lower batch sizes increasing the training time, we opted to stick with a batch size of 64 for the following experiments.

7.4.2 User Encoders

We continued by testing the set of user encoders that we presented in Section 7.1.2, again using regular DQN. We used the set of hyperparameter values that yielded the best results in the experiments described in the previous subsection. Just like before, we used the embedding of the concatenated title and abstract as the news encoding. Both the RL hyperparameters, and the news encoder scheme remained fixed for these experiments. As already discussed in Section 7.1.2, we used a reading history length of 40 articles for those user encodings that required a predetermined history length.

The following enumeration discusses the results of each graphic in Figure 7.5, where the item label corresponds to the respective caption label:

(a) For the weighted mean encoding, we tested various reading history lengths, i.e. amounts of news articles that are included in a user's history, namely the last 5, 20 and 40 articles. Furthermore, we also tested the performance of including all previously read articles in the encoding. The results show that including more articles in the user encoding improves preference elicitation. There even is a noticeable improvement from using 40 news items to processing the entire history, despite only approximately 10% of users having histories of more than 40 items (see Section 6.2.3). The significant performance drop between using 20 and 5 articles underlines that short-term interests are not sufficient to construct an accurate model of a user's preferences.

(b) We tested the effect of changing, or completely omitting, the weighting factor alpha. Applying no weighting ($\alpha = 1$) means that the reading order is completely disregarded, whereas decreasing α reduces the contribution of articles read in the distant past. As one would expect, applying no weighting negatively impacts performance, although the final performance is quite close. The comparison of $\alpha = 0.99$ and $\alpha = 0.999$ shows that a weighting factor closer to 1, i.e. one that decays more slowly and thus incorporates older articles more strongly into the weighted mean, performs slightly better. However, the difference is negligible.

(c) The multi-head self-attention encoding scheme works slightly better when using an attention pooling method, as opposed to a mean pooling, albeit the latter performs better at the early checkpoints, presumably due to the additional parameters that need to be trained for additive attention.

(d) We tested the effect of omitting and applying weighting ($\alpha = 0.999$) on the LTSTL encoding method, finding that the impact was minimal.

(e) We did not test any parameter configurations for the distribution encoding.

(f) Analogously to the multi-head self-attention encoding, the bi-directional GRU encoding using mean pooling learns faster, but is ultimately outperformed by attention pooling, with marginal differences. Utilizing just the last hidden state, as

opposed to pooling all of them together, strongly underperforms, indicating that it does not hold sufficient information to adequately represent the entire reading history. There are several potential reasons for this. For example, the range of the dependencies in the sequence might be too long to be entirely captured by the last hidden state. Another reason could be that this scheme focuses too much on the most recently read articles, which might be less relevant than the overall reading habits of the users.

(g) Finally, we compare all five distinct user encoding methods, with their respective best parameter settings. GRU performs best, followed by the weighted mean, self-attention, distribution encoding and in last place LTSTL. To us, the quality of the weighted mean encoding was surprising, due to its crude nature in comparison with the others. A possible explanation is that it strikes a good balance between encompassing a user's interests, with a focus on recently consumed news, while already generalizing at the user encoding stage. A key advantage of this encoding method is its lightweight nature. In contrast, while GRU outperforms it, the increase in trainable parameters comes with a significantly increased training time. In our opinion, the lackluster results of LTSTL show that the most recently read articles bear less valuable information than perhaps expected.

7.4.3 News encoders

Analogously, we tested the set of news encoders that we presented in Section 7.1.1, with regular DQN and the same hyperparameters. However, as opposed to using the best-performing user encoder, i.e. GRU with attention pooling, we opted for the weighted mean, due to its much lower training time.

The following enumeration discusses the results of each graphic in Figure 7.6, where the item label corresponds to the respective caption label:

(a) We started by comparing the performance of using the embedding of the title, the concatenation of the separate embeddings of title and abstract, as well as the embedding of the concatenation of title and abstract (which we have been using in all previous experiments). Unsurprisingly, the embedding of the concatenated texts achieved the best performance. Using both texts improves the PLM's understanding of an article's context, thus generating better embeddings. For instance, an article about basketball could only mention "the Bulls" in the title, whereas the abstract might contain additional information that could allow the PLM to infer that the title does not refer to the animals, but to the basketball team from Chicago. Nonetheless, using both title and abstract embeddings separately still performed better than relying on just the title or abstract. Out of the two, utilizing just the title was better than using just the abstract, despite titles being generally shorter than abstracts, see Figure 6.3. We can infer that users focus on the title, whereas the abstract is useful for our models to understand the context of the news item and

produce a good embedding. Furthermore, as discussed in Section 6.3, it is unclear whether the abstract was visible to the user before clicking on the item.

(b) In addition to the embedding of title and abstract, we included information about the news item's category. We tested the embedding of the category's name, as well as a one-hot encoding over all 15 categories. Clearly, a one-hot encoding outperformed the text embedding at every training step. However, note that omitting the category entirely was even better.

(c) We continued by adding the embedding of the subcategory text to the input. Due to the large amount of subcategories, and their dependence on the overarching category, we did not test a one-hot encoding. Just like with the category, performance decreased. Standardizing the input to a mean of 0 and a standard deviation of 1 did not help much, but reduced the standard deviation of the results.

(d) The embeddings of the concatenation of all textual news item elements, i.e. category, subcategory, title and abstract, into a single string also did not outperform the encodings that omitted the categories entirely. In the mean, it performed better than the encoding using the subcategory embedding, presumably due the very volatile results of the latter news encoder over the two random seeds, as can be seen from the standard deviation in the figure. The figure clearly shows that each additional input element beyond the concatenation of title and abstract led to a performance deterioration, with the complete concatenation embeddings achieving the lowest results (with the possible exception of the subcategory encoding, where more experiments would be needed to reduce the standard deviation of the results). There are several possible reasons for why this is the case. Firstly, the additional information could be redundant, while simultaneously occupying a large portion of the input vector (see Figure 7.1), hindering the LDNN from focusing on useful information. Secondly, our LDNN might lack the capacity, i.e. the network not being wide and/or deep enough, to extract useful patterns from the additional information. Thirdly, the enlarged feature space might hinder the LDNN from generalizing, causing it to overfit to the training data instead. Last but not least, while not having done a formal investigation into this issue, our manual exploration of the MIND dataset has revealed many mislabeled categories, where the attached article did not accurately fit the description. In addition, many of the subcategories that we remapped to natural language descriptions during preprocessing (see Section 6.2.2) had the appearance of being auto-generated, as opposed to manually labeled, suggesting the use of a possibly error-prone automated process. Ultimately, the benefit of using just the embeddings of the concatenated title and abstract is that each extension of the input vector also increases training time.

(e) The inclusion of numerical features regarding the respective news item also reduces performance. Omitting engagement features (see Section 6.2.4) yields better results. As discussed before, the information provided by the numeric features might be redundant and causing the network to overfit, or otherwise hinder its ability to

generalize. Therefore, it is plausible that the omission of some features leads to improvements.

(f) In order to gauge the impact of the embedding model, we tested a model by OpenAI, as discussed in Section 7.1.1. We generated embeddings for the concatenation of title and abstract. We hypothesized that a comparatively recently trained, pay-to-use model would improve performance. In addition, the dimension of the produced embeddings is twice as large as those returned by MPNet, thus carrying more information. However, MPNet significantly outperformed OpenAI's model, which we tested with two different learning rates. It is possible that the more compact embedding facilitates generalization. Another explanation is that, as discussed before, the LDNN does not have sufficient capacity to properly utilize the larger embeddings.

7.4.4 DRL algorithms

Last but not least, we tested the DRL algorithms that we have mentioned in Section 7.2, and theoretically discussed in Chapter 4. Again, we opted for the weighted mean user encoder. Given the results presented in the previous section, we used the embedding of title and abstract as a news encoding.

The following enumeration discusses the results of each graphic in Figure 7.7, where the item label corresponds to the respective caption label:

(a) We compared several target update schemes with DDQN, namely a direct update every 5k steps, as well as soft target updates with τ values of 0.005, 0.01 and 0.05. A soft target update with $\tau = 0.01$ performed best, followed by the direct target update. However, the differences were marginal.

(b) On average, DDQN improves on DQN, but the standard deviation is higher across different seeds, i.e. the results fluctuate more strongly, even at later checkpoints. In the following experiments, we opted to use the direct target update after 5k steps, for its seemingly better stability.

(c) In contrast to DDQN, the extension of DQN with a dueling network performed much worse than the regular version, despite the Atari benchmark results suggesting a different outcome. However, it should be noted that our approach of individually processing each candidate is not suitable for the default dueling architecture. Usually, one would use only the state as input, and compute the state value in one stream of the ANN, and the advantages for each action in a separate stream (see [87]). Subsequently, these two streams would be combined via an implementation of Equation 4.7. Due to our unbounded action space, which is a case that is rarely addressed in RL literature, leaving us without a precedent, we could not adopt this architecture. Refer to the code repository on GitHub[5] for our solution.

[5] https://github.com/d-vesely/drlnrs

(d) C51 outperformed DDQN. Notably, the results had a small standard deviation across different seeds and the algorithm learned significantly faster than DDQN, although the final results were close.

(e) As discussed in Section 4.3.2 on the theory behind QR-DQN, the number of quantiles is a hyperparameter that can be tuned. We tested two values suggested by the authors, namely 32 and 64. While the latter yielded better results with steadily improving scores at every checkpoint, both were outperformed by C51. Given that QR-DQN significantly improves on C51 when tested on the Atari benchmark, this result was surprising.

(f) Analogously, we tested IQN with 32 and 64 quantiles, obtaining the same result, whereby a higher amount of quantiles reaches better performance. However, in addition to the decreased performance of QR-DQN in comparison to C51, IQN obtained even lower scores. Using a soft target update minimally improved the results, but generally did not mitigate the issue. Again, this result is surprising, given the Atari benchmark results of these algorithms.

(g) Last but not least, we tested FQF, again with 32 and 64 quantiles. While it improved on IQN, its performance was disappointing, as can be seen in the next figure.

(h) Comparing all distributional DRL algorithms, the result is that C51 outperforms all others. This was unexpected, since each subsequent algorithm not only theoretically improved on its predecessor, but also in practice, achieving increasingly higher scores on the Atari benchmark. Furthermore, we note that regular DQN yields better results than its distributional counterparts, with the exception of C51. A possible explanation of C51's performance stems from the observation made by the authors of [6], which we have briefly mentioned at the end of Section 4.3.1. To reiterate, C51 achieved great results on Atari games with sparse rewards, namely "Venture" and "Private Eye", indicative of the ability of value distributions to propagate rare events better than DQN. As we have seen in Section 6.2.3, a user clicking on recommended items, and thus rewarding the agent, is indeed a rare event. The subsequently published QR-DQN, IQN and FQF all outperformed C51 (and each other) on the overall Atari benchmark, i.e. on an average measure over all Atari games. However, C51 kept the highest score on the aforementioned games, excluding the human score on "Private Eye" (see Atari score tables in [6, 19, 20, 94]). While this serves as a possible explanation for C51 outperforming its successors, it does not illuminate, why DQN also outperformed them.

In regards to REINFORCE, we do not present any results here. While the algorithm worked, it learned slowly and never achieved results comparable to those presented previously, albeit beating the baseline. Specifically, a REINFORCE agent trained on 300k episodes, which has a training time that is comparable to the experiments discussed so far, reaches a discounted return of 0.7338. While it is possible that REINFORCE

would reach scores similar to those presented, given more training time, we did not deem it worthwhile.

Finally, we tested both of the continuous action approaches that we have outlined in Section 7.1.3, i.e. outputting a scoring vector (indirect) and outputting a prototype-action (direct, with $k = 25\%$), with DDPG and TD3. In both cases, a common problem emerges that we have discussed in Section 4.5, namely the actor outplaying the critic. Relatively early on, the critic starts attaching higher and higher values to the actor's outputs, beyond reasonable boundaries. This overestimation spiral cannot be corrected, leading to the actor producing increasingly useless values, albeit receiving extremely high action-values. This problem occurs for both the indirect and the direct approach. Furthermore, using TD3, which purposely includes mechanisms to avoid such an exploitation issue, does not solve the problem and ultimately leads to the same results. However, the actor's output usually explodes in magnitude. In the case of the direct approach, the prototype-action vector should have the same properties as the known embedding vectors, e.g. regarding the sum of its squared components or the value range. Therefore, we tested incorporating a loss component into the actor that penalizes it, when the sum of squared vector components diverges from the hardcoded value. This value is known, since it can be obtained from samples of the embeddings produced by the news encoder. Indeed, this prevents the actor from spiraling out of control. We tried two evaluation approaches, utilizing just the actor, as well as the critic respectively. The former simply ranks the candidate items by their similarity to the prototype-action. The latter uses the prototype-action to reduce the number of candidates by half ($k = 50\%$), leaving similar candidates to be ranked by the critic, while the rest is again ranked by similarity. We did not further pursue the indirect approach.

The results are displayed in Figure 7.8. It is important to reemphasize that the actors do not use the list of candidate items, only the reading history. Therefore, while the results seem underwhelming at first, they show that an agent that uses just an actor significantly outperforms the baseline, without incorporating available news items. Such an agent could thus be used in a first step, to reduce a set of candidates that is too large to be processed in its entirety. In addition, using a critic to rank a selected subset further improves performance significantly. Nevertheless, to test the viability of this approach, additional data is required that includes information about the candidate set construction. TD3 (also in Figure 7.8) performs similarly, with the actor-critic evaluation being slightly better, and the actor-only evaluation slightly worse than DDPG.

7.4.5 Sequentiality

Due to its significantly increased evaluation time, we have tested the sequential evaluation approach described in Section 7.3 just once. It led to a decrease in performance, albeit marginal. Concretely, one run of the DQN agent visualized in Figure 7.6(a) scored 0.7828, whereas the sequential evaluation yielded a score of 0.7819. This could further indicate that the most recently read articles have little impact on a user's decision making. However, as already discussed in Section 6.3, it must be kept in mind that

impressions were shuffled in the MIND dataset. Therefore, our agent can have difficulties understanding the notion of ignoring an article, because a similar one was already read. For the same reason, the click labels are inherently unreliable, because a click might have not occurred due to the recommendation order, and potentially should have occurred in a ranking produced by our agent. Only an online study could provide clarity on these issues.

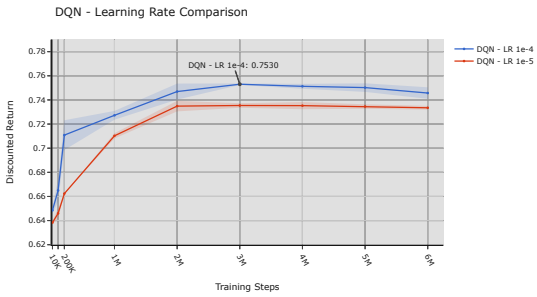
7.4.6 MIND Leaderboard

In order to compare the results of a DRLNRS to other approaches, especially those presented in Section 5.2, we submitted the test set predictions of the best-performing DRL algorithm, i.e. C51, to the MIND dataset's testbed[6] (see Section 6.1). We used the model trained with one of the two seeds, without performing any additional hyperparameter tuning. Furthermore, note that with a PMP of 30% and a training time of 6MM steps, our agent was statistically trained on less than half of the available positive experiences (see Section 7.1.5). Therefore, despite some experiment results decreasing at later checkpoints, it is still possible that a longer training, perhaps in conjunction with an adjusted learning rate decay, could improve performance. While the leaderboard[7] does not seem to receive updates any longer, we were still able to obtain result scores, see Table 7.3. Due to the iterative development and improvement of the SOTA methods, increasingly better scores are reported across several papers. In general, we conclude that a DRL approach, represented by C51, outperforms LSTUR and NRMS in their initial form, where they score an AUC of 0.6708 and 0.6766 respectively. However, as we discussed, the addition of a PLM that is fine-tuned during training significantly enhances these methods. Note that we used a PLM to produce embeddings, but we did not fine-tune the PLM during training. Concretely, LSTUR and NRMS with BERT achieve an AUC of 0.6949 and 0.6950 respectively, using UniLM [24] instead of BERT raises these scores to 0.7056 and 0.7064. UNBERT and MINER, both of which utilize BERT in their default version, achieve an AUC of 0.7068 and 0.7151 respectively. Comprehensive score lists are best found in [48] and [88]. Furthermore, note that ensembles of multiple independently trained models can further boost performance [48, 89, 96].

	AUC	MRR	nDCG@5	nDCG@10
C51	0.6828	0.3349	0.3654	0.4227
Fastformer	0.7304	0.3770	0.4180	0.4718

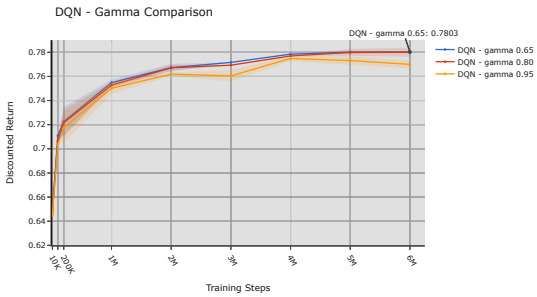
Table 7.3: MIND competition result scores for C51 and the current competition leader, a Fastformer variant.

[6] https://codalab.lisn.upsaclay.fr/competitions/420
[7] https://msnews.github.io/#leaderboard

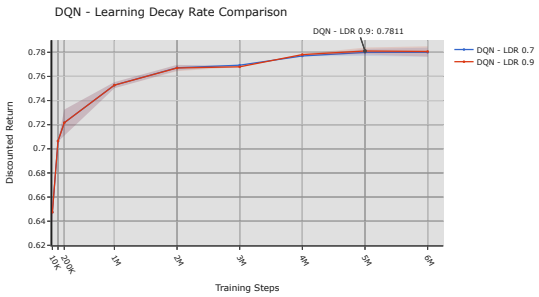


(a) Effect of learning rate.

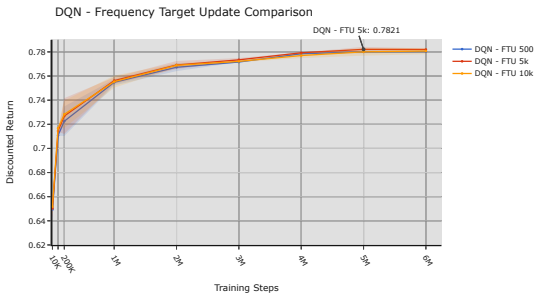
(b) Effect of positive memory preference.



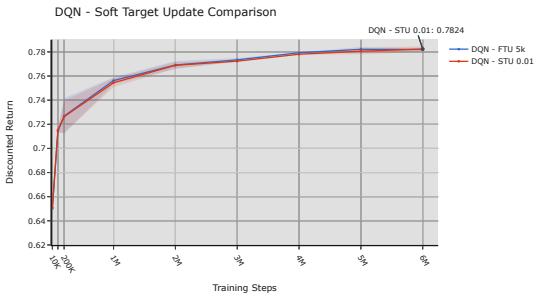
(c) Effect of discount value γ.



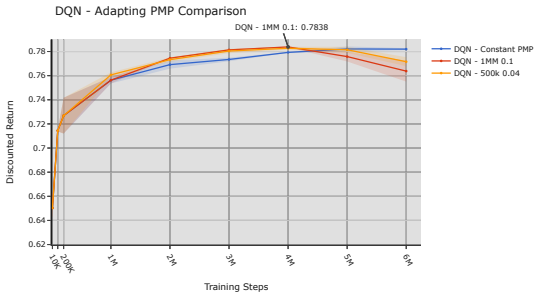
(d) Effect of learning decay rate.



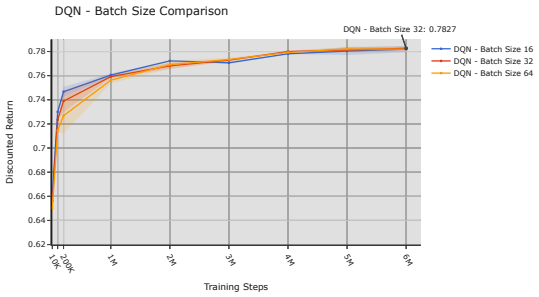
(e) Effect of target update frequency.



(f) Effect of soft target update.

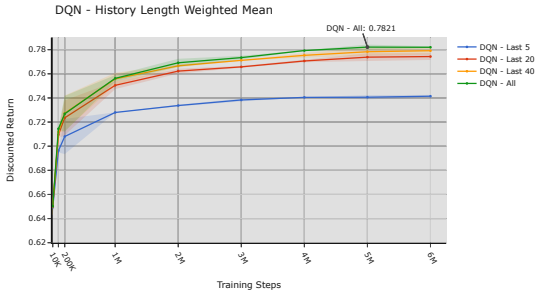


(g) Effect of adaptive pos. memory preference.

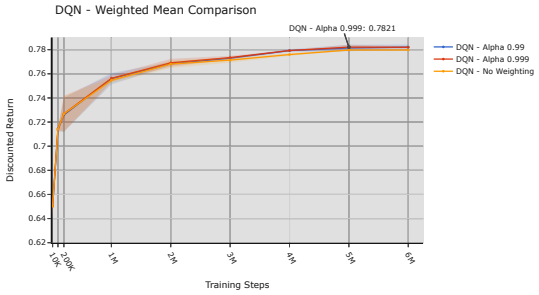


(h) Effect of batch size.

Figure 7.4: Result of varying RL hyperparameters, tested with DQN.

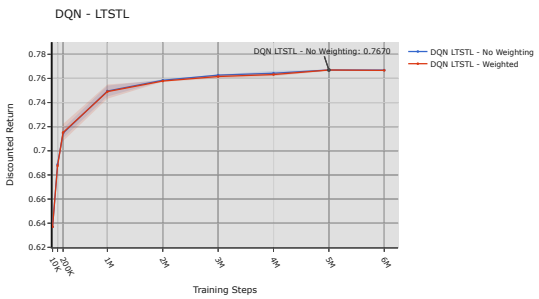


(a) Weighted mean encoding - history length.

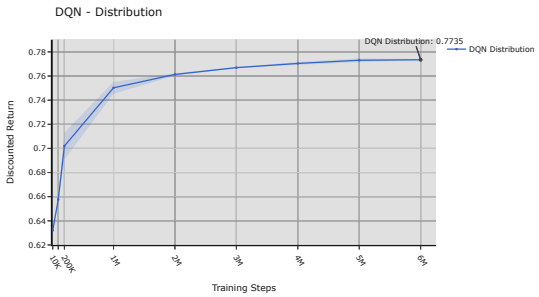


(b) Weighted mean encoding - alpha.

(c) Attention encoding.

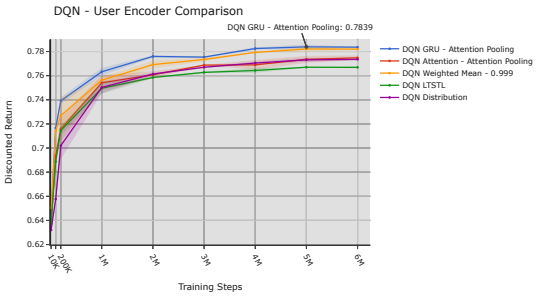


(d) LTSTL encoding.



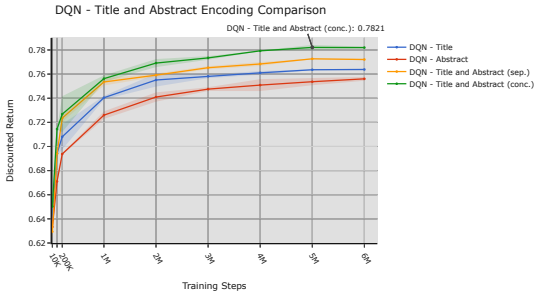
(e) Distribution encoding.

(f) GRU encoding.

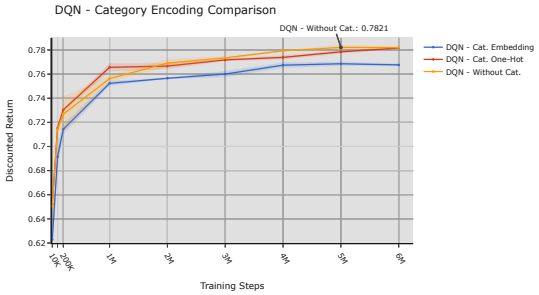


(g) User encoding comparison.

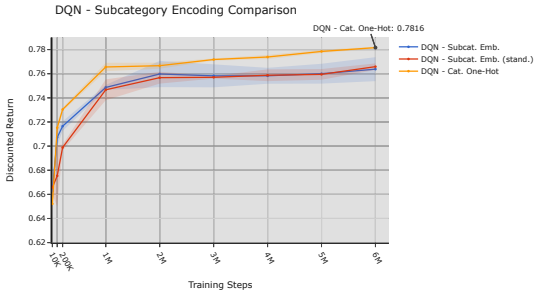
Figure 7.5: Results of varying user encodings, tested with DQN.



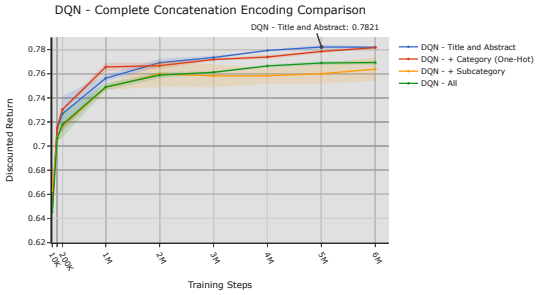
(a) Title and abstract encoding.



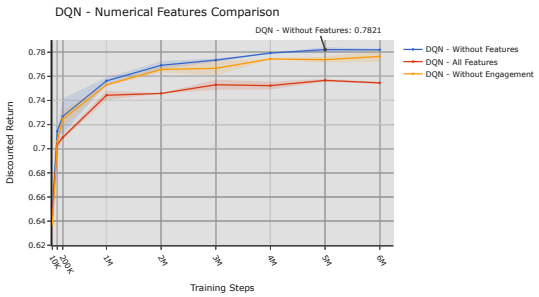
(b) Category encoding.



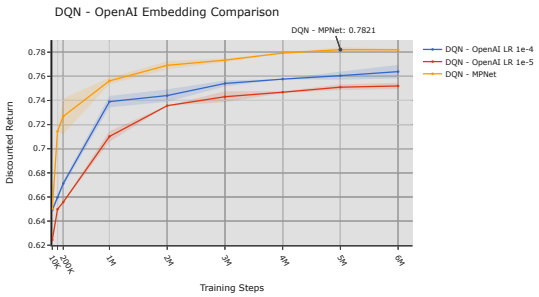
(c) Subcategory encoding.



(d) Complete concatenation encoding.

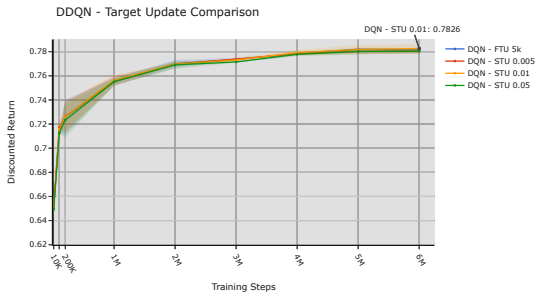


(e) Numeric features comparison.

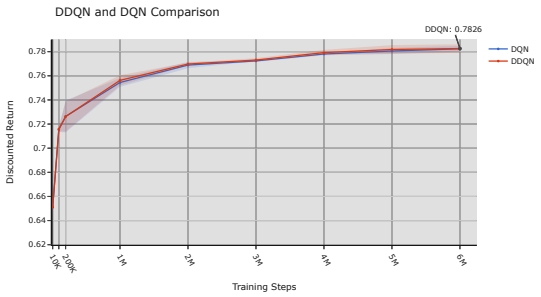


(f) Embedding model comparison.

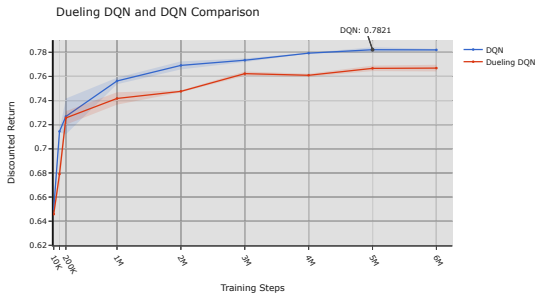
Figure 7.6: Results of varying news encodings, tested with DQN.



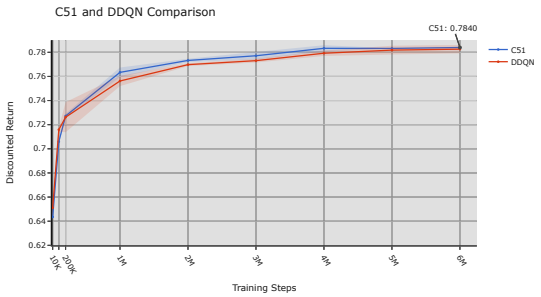
(a) DDQN target update comparison.



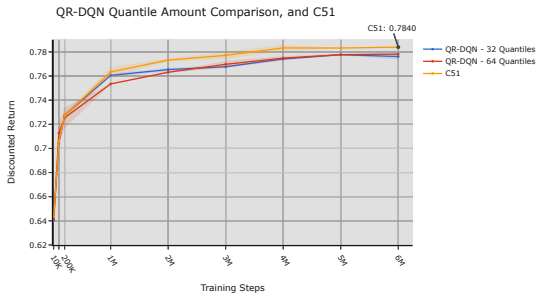
(b) DDQN and DQN.



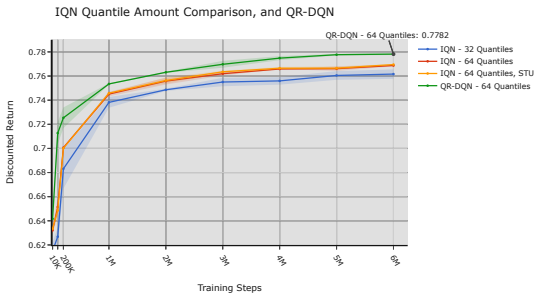
(c) Dueling DQN and DQN.



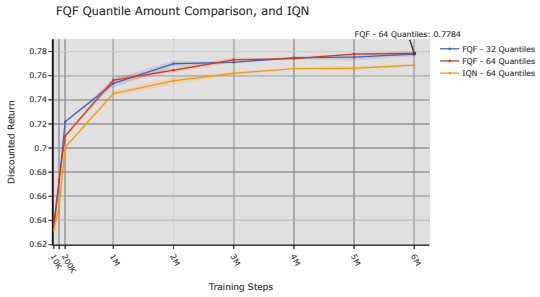
(d) C51 and DDQN.



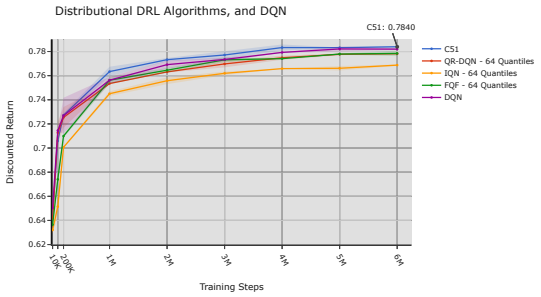
(e) QR-DQN quantile amounts, and C51.



(f) IQN quantile amounts, and QR-DQN.

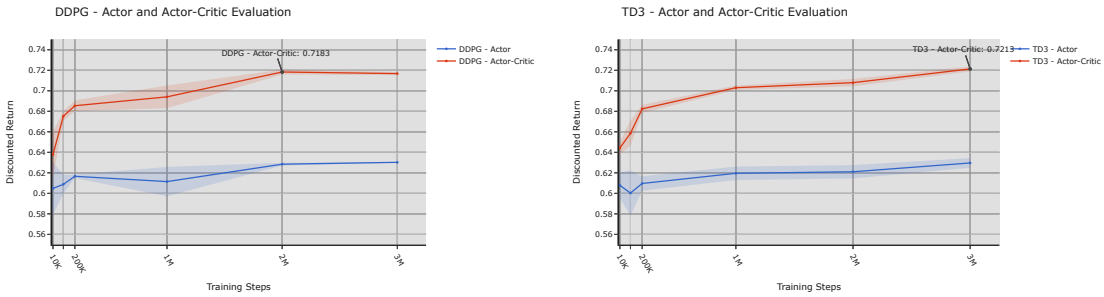


(g) FQF quantile amounts, and IQN.



(h) Comparison of Distributional DRL.

Figure 7.7: Results of comparative analysis of DRL algorithms.



(a) DDPG actor and actor-critic results.

(b) TD3 actor and actor-critic results.

Figure 7.8: DDPG and TD3 results.

CHAPTER 8

Conclusion

In this final chapter, we will summarize the findings and contributions of this thesis, as well as succinctly answer the research questions posed in Section 1.2. Finally, we want to offer brief suggestions for future work that would, in our opinion, drive the research on DRL(N)RSs forward.

8.1 Insights

The following bullet points summarize the key insights that we have gained from this thesis, and concretely the comparative performance analysis, most of which we have already discussed in the Section 7.4 as part of the results presentation.

- *Learning Parameters*: Aside from the learning rate, the hyperparameter with the largest impact was the positive memory preference, which guides oversampling. Its strong effect on an agent's performance suggests that it is a highly important factor when training RL agents with a replay memory constructed from an offline dataset, as we did, and should be carefully tuned. Using an adaptive positive memory preference increased the training speed, but performance ultimately dipped below the levels achieved when keeping a constant rate. Last but not least, a discount factor that was too high led to a decrease in performance, because the agent is punished less for ranking positive items lower. Otherwise, training was relatively insensitive to other hyperparameters.

- *User Encoding*: A simple weighted mean over a user's history almost performed best, being just barely surpassed by the user encoders relying on a bi-directional GRU. However, the large amount of additional training parameters severely increased the training time as well. The close results of those two methods, as well as the low scores reached when using just the last hidden state of the GRU, instead of

pooling all of them together, raises questions about the importance of the order of a user's reading history. In other words, the sequence of previously read articles, and the concrete news items that were read most recently, might have less impact on a user's next choices than expected. The authors of [90] raise the same question, coming to the conclusion that modeling news recommendation as a conventional sequential recommendation problem is sub-optimal. Nonetheless, we have shown that a weighted mean does outperform an unweighted mean. At the same time, processing the entire histories yielded better results than reducing histories to exclude articles that were not recently read. These two results indicate that it is important to differentiate between long-term and short-term interests of a user, without focusing too much on short-term dependencies between specific news items, that might not be as strong, as short-term, or as sequential, as one might think. Our sequential evaluation approach, discussed in Section 7.4.5, underlines this sentiment. Other encoding methods did not reach competitive levels, with the LTSTL encoding achieving the lowest scores. Note that LTSTL includes the encoding of the most recently read news item in the user encoding. Yet again, we must emphasize the need for an online study, to obtain truly conclusive results on this issue.

- *News Encoding*: An embedding of the concatenated title and abstract performed best. Additional information, either in the form of category and subcategory, or as numerical features, led to a decreased performance. Potential reasons are that the information was redundant, that it hindered generalization, or a lack of capacity in our LDNNs. Using just the title or abstract was insufficient, with the former outperforming the latter.

- *DRL Algorithms*: C51 achieved the highest performance out of the tested DRL algorithms, followed by DDQN and regular DQN. DDQN did outperform C51 at one of the two tested seeds. However, due to DDQN's results having a high standard deviation, it ended up being slightly worse on average. In contrast, C51 achieved stable results across both seeds and learned faster, reaching good results at early checkpoints. C51's successor algorithms in the category of distributional RL, i.e. QR-DQN, IQN and FQF, all performed worse than C51. This supports the theory that C51 operates well in sparse reward environments. As already discussed, C51 scored higher on the sparse reward Atari games "Private Eye" and "Venture", than its successor algorithms. However, we have no concrete explanation for why DQN was able to score higher in our experiments, than the distributional algorithms that convincingly beat DQN on the Atari benchmark, including the two aforementioned games.

- *Candidate Set Extraction*: The results achieved by DDPG/TD3 show that agents can outperform the baseline without having to process the set of candidate items, thus presenting a possible approach to reducing very large item sets to a small set of candidate items. However, the viability of this approach would require further testing that cannot be conducted with the available data.

8.2 Summary

The contributions of this thesis are best summarized by outlining, what the research field was missing. Firstly, as discussed in Section 5.3, little DRLRS research is conducted in the news domain ([99] and [50]). Secondly, the majority of DRLRS experiments are conducted on proprietary data, rendering them not reproducible. Thirdly, DRLRS research is usually focused on a specific algorithm, without drawing comparisons to systems based on different DRL algorithms. In our opinion, such an approach clearly diminishes the insights on the general viability of combining DRL and RSs. Our thesis was aimed at filling these spaces, and we believe to have succeeded, by providing a comprehensive comparative performance analysis of several DRLNRSs, using the publicly available MIND dataset [93]. In addition to testing DRL algorithms, we also compare different user encoders and news encoders, as opposed to focusing on any specific settings, which ensures the generality and robustness of our results, and provides further insights. We published all of our code and experiment configurations on GitHub[1], making our results easily reproducible. Our codebase requires minimal dependencies and does not involve any third-party RL libraries or frameworks, with all algorithms coded from scratch. This further increases reproducibility and transparency, as well as facilitates the adaptation of our code. In summary, our analysis offers insights into the overall viability of applying DRL to recommendation problems, and the performance of individual DRL algorithms, in conjunction with various user- and news encoders. Furthermore, along with our code, this thesis presents, discusses and provides a framework for training DRLRSs, which can be reused in other recommendation domains. Last but not least, our thesis summarizes all necessary background knowledge to understand the thesis and the SOTA, and we believe it can serve as a starting point for delving into the topic, and for further research into DRLNRSs.

Finally, we have answered the research questions posed in Section 1.2. Mainly, the answers can be found in Section 7.1 (**Q1**), 7.1.5 (**Q2**), 7.4.4 (**Q3**), 7.4.3 and 7.4.2 (**Q4**), 7.4.6 (**Q5**), 8.1 and 7.4.5 (**Q6**).

8.3 Future Work

The following list contains a few brief directions for future work that would benefit the research field of deep reinforcement learning recommender systems, in the news domain and others:

- *Dataset*: The MIND dataset was undoubtedly very useful, especially since no other comparable datasets exist as of August 2023. However, we already outlined some criticisms in Section 6.3. To us, the key issue was the shuffled order of impressions. It does not accurately reflect the user's reading habits and could hinder the training of RL agents. Simply speaking: if the agent does not know, whether article A was

[1] https://github.com/d-vesely/drlnrs

read before article B was skipped, then it cannot differentiate whether the user skipped B because it was uninteresting, or because it was perceived to contain the same information as A. Therefore, a dataset that preserves the sequential order of events would be highly useful for DRLRS research, and further clarify the questionable role of sequentiality in NRSs. The same is the case for a dataset that (reliably) includes information on the publisher, author and date of an article.

- *Online Experiments*: Naturally, conducting online experiments in a live RS environment, as discussed in Section 3.3, would be extremely beneficial to the training and evaluation of RSs based on DRL. While the results obtained in this thesis are certainly insightful, DRL agents have the potential to thrive in an online environment. Furthermore, the labels are predetermined in an offline evaluation, but the actual order of recommendations could have a large impact on the user's decisions. For instance, if a user is recommended two articles on the same topic, they might read the first recommendation and skip the second one. Therefore, if the RS that produced the labels offered a different order than our RL agent, our result scores will decrease, despite giving an equally good or better recommendation.
- *Sequentiality*: Further research should be conducted on the importance of sequentiality in NRSs.
- *Beyond Accuracy*: Further research should be conducted into the topic of evaluating NRSs beyond accuracy. Furthermore, datasets that facilitate these kinds of evaluation metrics should be constructed.
- *DRL Algorithms and Encoders*: There is a multitude of other DRL algorithms that could be applied to the news recommendation problem. The same is the case for news- and user encoders.
- *RL Tooling*: In contrast to SL, RL tooling and standardization is - to put it bluntly - a mess. The multitude of incomplete, incompatible, and little maintained libraries hinders productivity and reproducibility. This is especially the case, when one is not operating in a pre-existing or easily implementable OpenAI gym[2] environment. A lot of work is required to raise the level of RL tooling, in terms of applicability and standardization, towards that of SL. Thankfully, the recently founded Farama[3] foundation seems to be pushing in that direction.

[2] https://gymnasium.farama.org/
[3] https://farama.org/Announcing-The-Farama-Foundation

List of Figures

List of Tables

List of Algorithms

Acronyms

AC Actor-Critic. 49

AI Artificial Intelligence. 5

ANN Artificial Neural Network. 7

C51 Categorical 51. 41

CNN Convolutional Neural Network. 55

DDPG Deep Deterministic Policy Gradient. 49

DDQN Double DQN. 37

DL Deep Learning. 7

DP Dynamic Programming. 16

DPG Deterministic Policy Gradient. 49

DQN Deep Q-Network. 35

DRL Deep Reinforcement Learning. 8

DRLNRS Deep Reinforcement Learning News Recommender System. 60

DRLRS Deep Reinforcement Learning Recommender System. 51

FPN Fraction Proposal Network. 47

FQF Fully Parameterized Quantile Function. 47

GPI Generalized Policy Iteration. 18

GRU Gated Recurrent Unit (Network). 56

IQN Implicit Quantile Networks. 45

LDNN Large Deep Neural Network. 8

LSTM Long Short-Term Memory. 56

MC Monte Carlo. 19

MDP Markov Decision Process. 10

MIND **MI**crosoft **N**ews **D**ata. 54

ML Machine Learning. 6

NRS News Recommender System. 51

PLM Pre-Trained Language Model. 57

QR-DQN Quantile Regression DQN. 43

QVN Quantile Value Network. 47

RL Reinforcement Learning. 5

RS Recommender System. 25

SL Supervised Learning. 6

SOTA State-Of-The-Art. 51

TD Temporal-Difference Learning. 21

TD3 Twin Delayed DDPG. 50

UL Unsupervised Learning. 7

Bibliography

[1] *"Reinforcement Learning for Recommender Systems: A Case Study on Youtube," by Minmin Chen*. Mar. 2019. URL: https://youtu.be/HEqQ2_1XRTs?t=50 (visited on 06/24/2023).

[2] M Mehdi Afsar, Trafford Crump, and Behrouz Far. "Reinforcement learning based recommender systems: A survey". In: *ACM Computing Surveys* 55.7 (2022), pp. 1–38.

[3] Charu C. Aggarwal. *Recommender Systems - The Textbook*. Springer, 2016.

[4] Mingxiao An et al. "Neural News Recommendation with Long- and Short-term User Representations". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, July 2019, pp. 336–345.

[5] Michael A. Beam. "Automating the News: How Personalized News Recommender System Design Choices Impact News Reception". In: *Communication Research* 41.8 (2014), pp. 1019–1041.

[6] Marc G. Bellemare, Will Dabney, and Rémi Munos. "A Distributional Perspective on Reinforcement Learning". In: *CoRR* abs/1707.06887 (2017).

[7] Marc G. Bellemare, Will Dabney, and Mark Rowland. *Distributional Reinforcement Learning*. http://www.distributional-rl.org. MIT Press, 2023.

[8] Richard Bellman. *Dynamic Programming*. 1st ed. Princeton University Press, 1957.

[9] Yoshua Bengio and Yann Lecun. "Scaling learning algorithms towards AI". In: *Large-scale kernel machines*. MIT Press, 2007.

[10] Balázs Bodó. "Selling News to Audiences – A Qualitative Inquiry into the Emerging Logics of Algorithmic News Personalization in European Quality News Media". In: *Digital Journalism* 7.8 (2019), pp. 1054–1075.

[11] Robin Burke. "Hybrid Web Recommender Systems". In: vol. 4321. Jan. 2007.

[12] Matt Carlson. "Automating judgment? Algorithmic judgment, news knowledge, and journalistic professionalism". In: *New Media & Society* 20.5 (2018), pp. 1755–1772.

[13] Netflix Help Center. *How does Netflix make recommendations for me?* n.d. URL: https://help.netflix.com/en/node/100639#:~:text=We%20estimate%20the%20likelihood%20that,preferences%20on%20our%20service%2C%20and (visited on 05/29/2023).

[14] Pew Research Center. *More than eight-in-ten Americans get news from digital devices*. Jan. 2021. URL: https://www.pewresearch.org/short-reads/2021/01/12/more-than-eight-in-ten-americans-get-news-from-digital-devices/ (visited on 06/06/2023).

[15] 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*. WSDM '19. Melbourne VIC, Australia: Association for Computing Machinery, 2019, pp. 456–464.

[16] Shi-Yong Chen et al. "Stabilizing Reinforcement Learning in Dynamic Environment with Application to Online Recommendation". In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '18. London, United Kingdom: Association for Computing Machinery, 2018, pp. 1187–1196.

[17] Xiaocong Chen et al. "A survey of deep reinforcement learning in recommender systems: A systematic review and future directions". In: *arXiv preprint arXiv:2109.03540* (2021).

[18] Huige Cheng. *MIND News Recommendation Technical Report*. Sept. 2020. URL: https://msnews.github.io/assets/doc/1.pdf (visited on 06/23/2023).

[19] Will Dabney et al. "Distributional Reinforcement Learning with Quantile Regression". In: *CoRR* abs/1710.10044 (2017).

[20] Will Dabney et al. "Implicit Quantile Networks for Distributional Reinforcement Learning". In: *CoRR* abs/1806.06923 (2018).

[21] Stephen Dankwa and Wenfeng Zheng. "Twin-Delayed DDPG: A Deep Reinforcement Learning Technique to Model a Continuous Movement of an Intelligent Robot Agent". In: *Proceedings of the 3rd International Conference on Vision, Image and Signal Processing*. ICVISP 2019. Vancouver, BC, Canada: Association for Computing Machinery, 2020.

[22] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, June 2019, pp. 4171–4186.

[23] Hao Dong et al. *Deep Reinforcement Learning: Fundamentals, Research, and Applications*. http://www.deepreinforcementlearningbook.org. Springer Nature, 2020.

[24] Li Dong et al. “Unified Language Model Pre-training for Natural Language Understanding and Generation”. In: *ArXiv* abs/1905.03197 (2019).

[25] Gabriel Dulac-Arnold et al. “Deep Reinforcement Learning in Large Discrete Action Spaces”. In: (2016).

[26] Stefan Ellerbeck. *Most people get their news online - but many are switching off altogether. Here's why*. Sept. 2022. URL: https://www.weforum.org/agenda/2022/09/news-online-europe-social-media/ (visited on 06/06/2023).

[27] Chong Feng et al. “News Recommendation Systems - Accomplishments, Challenges & Future Directions”. In: *IEEE Access* 8 (2020), pp. 16702–16725.

[28] Sharon Ferguson. *How Does Spotify Know What You Like? Expert Sheds Light on Recommender Systems at U of T Event*. May 2023. URL: https://www.utoronto.ca/news/how-does-spotify-know-what-you-expert-sheds-light-recommender-systems-u-t-event (visited on 05/29/2023).

[29] Sharon Ferguson. *Powered by AI: Instagram's Explore Recommender System*. Nov. 2019. URL: https://ai.facebook.com/blog/powered-by-ai-instagrams-explore-recommender-system/ (visited on 05/29/2023).

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

[31] Ryan Greene et al. *New and improved embedding model*. Dec. 2022. URL: https://openai.com/blog/new-and-improved-embedding-model (visited on 06/21/2023).

[32] Gustaf Gripenberg. “Approximation by neural networks with a bounded number of nodes at each level”. In: *Journal of Approximation Theory* 122.2 (2003), pp. 260–266.

[33] Larry Hardesty. *The History of Amazon's Recommendation Algorithm*. Nov. 2019. URL: https://www.amazon.science/the-history-of-amazons-recommendation-algorithm (visited on 05/29/2023).

[34] Mance E. Harmon, Leemon C. Baird, and A. Harry Klopf. “Advantage Updating Applied to a Differrential Game”. In: *NIPS*. 1994.

[35] Mark Harmon. “Multi-player residual advantage learning with general function”. In: 1996.

[36] Hado Hasselt. “Double Q-learning”. In: *Advances in Neural Information Processing Systems*. Vol. 23. Curran Associates, Inc., 2010.

[37] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-learning”. In: *CoRR* abs/1509.06461 (2015).

[38] Ji He et al. “Deep Reinforcement Learning with an Unbounded Action Space”. In: *ArXiv* abs/1511.04636 (2015).

[39] Natali Helberger. “On the Democratic Role of News Recommenders”. In: *Digital Journalism* 7.8 (2019), pp. 993–1012.

[40] Matteo Hessel et al. “Rainbow: Combining Improvements in Deep Reinforcement Learning”. In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*. AAAI’18/IAAI’18/EAAI’18. New Orleans, Louisiana, USA: AAAI Press, 2018.

[41] G. E. Hinton, S. Osindero, and Y. W. Teh. “A Fast Learning Algorithm for Deep Belief Nets”. In: *Neural Computation* 18 (2006), pp. 1527–1554.

[42] T. F Hoad. *The Concise Oxford Dictionary of English Etymology*. Oxford University Press, 2002.

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

[44] Peter J. Huber. “Robust Estimation of a Location Parameter”. In: *The Annals of Mathematical Statistics* 35.1 (1964), pp. 73–101.

[45] Stratton C. Jaquette. “Markov Decision Processes with a New Optimality Criterion: Discrete Time”. In: *The Annals of Statistics* 1.3 (1973), pp. 496–505.

[46] Mozhgan Karimi, Dietmar Jannach, and Michael Jugovac. “News recommender systems – Survey and roads ahead”. In: *Information Processing & Management* 54.6 (2018), pp. 1203–1227.

[47] Roger Koenker. *Quantile Regression*. Cambridge University Press, 2005.

[48] Jian Li et al. “MINER: Multi-Interest Matching Network for News Recommendation”. In: *Findings of the Association for Computational Linguistics: ACL 2022*. Association for Computational Linguistics, May 2022, pp. 343–352.

[49] Timothy Lillicrap et al. “Continuous control with deep reinforcement learning”. In: *CoRR* (Sept. 2015).

[50] Danyang Liu et al. “Reinforced Anchor Knowledge Graph Generation for News Recommendation Reasoning”. In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. KDD ’21. Virtual Event, Singapore: Association for Computing Machinery, 2021, pp. 1055–1065.

[51] Feng Liu et al. “Deep Reinforcement Learning based Recommendation with Explicit User-Item Interactions Modeling”. In: *ArXiv* abs/1810.12027 (2018).

[52] Feng Liu et al. “End-to-End Deep Reinforcement Learning Based Recommendation with Supervised Embedding”. In: *Proceedings of the 13th International Conference on Web Search and Data Mining*. WSDM ’20. Houston, TX, USA: Association for Computing Machinery, 2020, pp. 384–392.

[53] Feng Liu et al. “State representation modeling for deep reinforcement learning based recommendation”. In: *Knowledge-Based Systems* 205 (2020).

[54] Andreas Lommatzsch, Benjamin Kille, and Sahin Albayrak. “Incorporating context and trends in news recommender systems”. In: *Proceedings of the International Conference on Web Intelligence* (2017).

[55] John McCarthy et al. “A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence, August 31, 1955”. In: *AI Magazine* 27.4 (Dec. 2006).

[56] Warren Mcculloch and Walter Pitts. “A Logical Calculus of Ideas Immanent in Nervous Activity”. In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 127–147.

[57] Eliza Mitova et al. “News recommender systems: a programmatic research review”. In: *Annals of the International Communication Association* 47.1 (2023), pp. 84–113.

[58] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518 (2015), pp. 529–533.

[59] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: (Dec. 2013).

[60] Judith Moeller et al. “Shrinking core? Exploring the differential agenda setting power of traditional and personalized news media”. In: *info* 18 (Sept. 2016), pp. 26–41.

[61] Miguel Morales. *Grokking Deep Reinforcement Learning*. Manning Publications Co., 2020.

[62] Nic Newman. *Overview and key findings of the 2022 Digital News Report*. June 2022. URL: https://reutersinstitute.politics.ox.ac.uk/digital-news-report/2022/dnr-executive-summary (visited on 06/06/2023).

[63] Özlem Özgöbek et al. “The 10th International Workshop on News Recommendation and Analytics (INRA 2022)”. In: July 2022, pp. 3470–3473.

[64] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2019.

[65] Jordan Pollack and Alan Blair. “Why did TD-Gammon Work?” In: *Advances in Neural Information Processing Systems*. Ed. by M.C. Mozer, M. Jordan, and T. Petsche. Vol. 9. MIT Press, 1996.

[66] Pascal Poupart. *CS885 Fall 2021 Reinforcement Learning - Module 5: Distributional RL*. University of Waterloo. 2021. URL: https://cs.uwaterloo.ca/~ppoupart/teaching/cs885-fall21/slides/cs885-module5.pdf (visited on 05/24/2023).

[67] Marc’Aurelio Ranzato et al. “Efficient Learning of Sparse Representations with an Energy-Based Model”. In: Jan. 2006.

[68] Shaina Raza and Chen Ding. “News recommender system: a review of recent progress, challenges, and opportunities”. In: *Artificial Intelligence Review* 55.1 (Jan. 2022), pp. 749–800.

[69] Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Nov. 2019.

[70] Francesco Ricci. “Recommender Systems: Models and Techniques”. In: *Encyclopedia of Social Network Analysis and Mining*. Springer New York, 2014, pp. 1511–1522.

[71] Francesco Ricci, Lior Rokach, and Bracha Shapira. *Recommender Systems Handbook*. Vol. 1-35. Oct. 2010.

[72] Martin Riedmiller. “Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method”. In: *Machine Learning: ECML 2005*. Springer Berlin Heidelberg, 2005, pp. 317–328.

[73] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Third Global Edition. Pearson, 2016.

[74] Claude Sammut and Geoffrey I. Webb. *Encyclopedia of Machine Learning*. 2nd. Springer Publishing Company, Incorporated, 2017.

[75] A. L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers”. In: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229.

[76] Tom Schaul et al. “Prioritized Experience Replay”. In: (Nov. 2015).

[77] Barry Schwartz. *The Paradox of Choice: Why More Is Less*. Harper Perennial, 2005.

[78] Guy Shani, David Heckerman, and Ronen I. Brafman. “An MDP-Based Recommender System”. In: *Journal of Machine Learning Research* 6.43 (2005), pp. 1265–1295.

[79] David Silver et al. “Deterministic Policy Gradient Algorithms”. In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 1. PMLR, 22–24 Jun 2014, pp. 387–395.

[80] Spotify. *About Spotify*. URL: https://newsroom.spotify.com/company-info/#:~:text=Discover%2C%20manage%20and%20share%20over,ad%2Dfree%20music%20listening%20experience. (visited on 06/05/2023).

[81] Statista. *Countries with most content available on Netflix worldwide as of March 2023*. 2023. URL: https://www.statista.com/statistics/1013571/netflix-library-size-worldwide/ (visited on 06/05/2023).

[82] Harald Steck et al. “Deep Learning for Recommender Systems: A Netflix Case Study”. In: *AI Magazine* 42.3 (Nov. 2021), pp. 7–18.

[83] Richard S. Sutton and Andrew G. Barton. *Reinforcement Learning: An Introduction*. 2nd ed. http://incompleteideas.net/book/the-book-2nd.html. MIT Press, 2018.

[84] Gerald Tesauro. “Temporal Difference Learning and TD-Gammon”. In: *Commun. ACM* 38.3 (Mar. 1995), pp. 58–68.

[85] J.N. Tsitsiklis and B. Van Roy. “An analysis of temporal-difference learning with function approximation”. In: *IEEE Transactions on Automatic Control* 42.5 (1997), pp. 674–690.

[86] Ashish Vaswani et al. “Attention Is All You Need”. In: *CoRR* abs/1706.03762 (2017).

[87] Ziyu Wang et al. *Dueling Network Architectures for Deep Reinforcement Learning*. 2016.

[88] Chuhan Wu et al. “Empowering News Recommendation with Pre-Trained Language Models”. In: *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’21. Virtual Event, Canada: Association for Computing Machinery, 2021, pp. 1652–1656.

[89] Chuhan Wu et al. “Fastformer: Additive Attention Can Be All You Need”. In: *ArXiv* abs/2108.09084 (2021).

[90] Chuhan Wu et al. “Is News Recommendation a Sequential Recommendation Task?” In: *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’22. Madrid, Spain: Association for Computing Machinery, 2022, pp. 2382–2386.

[91] Chuhan Wu et al. “Neural News Recommendation with Multi-Head Self-Attentio”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP*. Association for Computational Linguistic, Nov. 201, pp. 6389–639.

[92] Chuhan Wu et al. “Personalized News Recommendation: Methods and Challenges”. In: *ACM Trans. Inf. Syst.* 41.1 (Jan. 2023).

[93] Fangzhao Wu et al. “MIND: A Large-scale Dataset for News Recommendation”. In: Jan. 2020, pp. 3597–3606.

[94] Derek Yang et al. “Fully Parameterized Quantile Function for Distributional Reinforcement Learning”. In: *CoRR* abs/1911.02140 (2019).

[95] Alexander Zai and Brandon Brown. *Deep Reinforcement Learning in Action*. Manning Publications Co., 2020.

[96] Qi Zhang et al. “UNBERT: User-News Matching BERT for News Recommendation”. In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*. Ed. by Zhi-Hua Zhou. International Joint Conferences on Artificial Intelligence Organization, Aug. 2021, pp. 3356–3362.

[97] Xiangyu Zhao et al. “DEAR: Deep Reinforcement Learning for Online Advertising Impression in Recommender Systems”. In: *AAAI Conference on Artificial Intelligence*. 2019.

[98] Xiangyu Zhao et al. “Deep Reinforcement Learning for Page-Wise Recommendations”. In: *Proceedings of the 12th ACM Conference on Recommender Systems*. RecSys ’18. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2018, pp. 95–103.

[99] Guanjie Zheng et al. “DRN: A Deep Reinforcement Learning Framework for News Recommendation”. In: *Proceedings of the 2018 World Wide Web Conference*. WWW ’18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 167–176.