
Design and Implementation of a Recommender System for News in ZDFMEDIATHEK based on Deep Reinforcement Learning

Konzeption und Implementierung eines Empfehlungsverfahrens für Nachrichten in der ZDFMEDIATHEK auf Basis des tiefen verstärkenden Lernens
Master-Thesis von Valentin Kuhn aus Frankfurt am Main
18. November 2019



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Design and Implementation of a Recommender System for News in ZDFMEDIATHEK based on Deep Reinforcement Learning

Konzeption und Implementierung eines Empfehlungsverfahrens für Nachrichten in der ZDFMEDIATHEK auf Basis des tiefen verstärkenden Lernens

Vorgelegte Master-Thesis von Valentin Kuhn aus Frankfurt am Main

1. Gutachten: Prof. Dr. Johannes Fürnkranz

2. Gutachten: Tobias Joppen

Tag der Einreichung: 18. November 2019

Erklärung zur Abschlussarbeit gemäß § 23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Valentin Kuhn, die vorliegende Master-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Datum / Date:

Unterschrift / Signature:

(Valentin Kuhn)



Abstract

Tremendous supply of online content puts a strain on users by overburdening their capabilities to filter interesting content. Therefore, over-the-top providers recognized a growing need for personalized recommendations. Especially in fast-paced domains, such as news, up-to-date filtering is crucial for user satisfaction. The domain of news recommendation features two characteristics setting it apart from many other recommendation applications: a dynamically changing corpus of items and user preference, and, rewards not necessarily coupled to immediate click-through rates. Thus, we describe a novel approach recently introduced by Zheng et al. [60] that explicitly considers these two limitations of prior recommendation engines in a deep Q-learning approach with Dueling Bandit Gradient Descent for exploration. Furthermore, we adapt the application of the presented deep Q-network to ZDF_{MEDIATHEK}.



Contents

1	Introduction	1
2	Background	5
2.1	Prior Work	5
2.2	Preliminaries	7
2.2.1	Reinforcement Learning	7
2.2.2	Neural Networks & Deep Learning	9
2.2.3	Deep Reinforcement Learning	11
3	Technique: DDQN	13
3.1	Recommendation Pipeline	14
3.2	Model Architecture	14
3.3	Input Features	16
3.4	Network Layers	19
3.5	Loss Function	21
3.5.1	Categorical Cross-entropy Loss	21
3.5.2	Binary Cross-Entropy Loss	22
3.6	Reward	22
3.6.1	Offline Reward	22
3.6.2	Online Reward	23
3.7	Exploration	25
3.7.1	Epsilon-Greedy	25
3.7.2	Dueling Bandit Gradient Descent	25
4	Implementation	29
4.1	Frameworks	30
4.2	Environment	32
4.3	Agent	33
5	Evaluation	35
5.1	Experiment Setup	36
5.2	Hyperparameters	36
5.3	Data	37
5.4	Experiment Results	37
6	Related Work	41
7	Conclusion	45
	Acronyms	V
	References	VII



List of Figures

1	Recommendations in ZDFmediathek	1
2	Reinforcement Learning in Recommender Systems	2
3	Candidate Generation in Recommender Systems	3
4	Comparison of Contextual Multi-Armed Bandit and Markov Decision Process	6
5	Agent and Environment in the Markov Decision Process	7
6	Artificial Neuron in a Multilayer Perceptron	10
7	Recommender-User Interactions in the Markov Decision Process	11
8	Conceptual View on the Technique in MDP	13
9	Candidate Generation in ZDFmediathek	14
10	Comparison of Network Architectures	15
11	Dueling Deep Q-Network Architecture in ZDFmediathek	16
12	Model Architecture of the Double Dueling Deep Q-Network	18
13	Comparison of ReLU Activations	20
14	User Activeness Estimation for a User	24
15	Conceptual View on the Technique in MDP	29
16	Dockerfile preparing our Tensorflow docker image	30
17	Keras Summary of the Q-network	31
18	Building simulator memory	32
19	Conceptual View on the Simulator Environment	35
20	Training Metrics	38
21	Comparison of Deep Q-Network Architectures	41



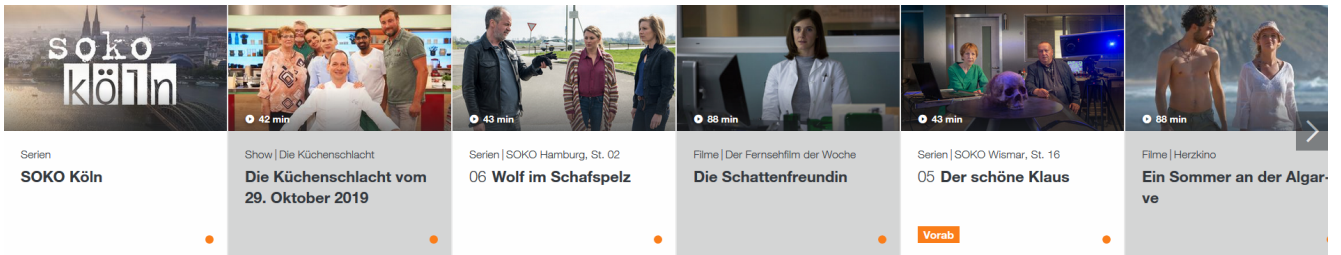


Figure 1: Recommendations in ZDFMEDIATHEK

1 Introduction

Recent improvements in communication technology and increasing user interests in non-linear streaming services have led to an enormous growth in online content, especially in Over-The-Top (OTT) media services. OTT services directly distribute content to users without intermediary distributors or platforms. For instance, Covington et al. [9] describe video recommendations out of a huge corpus for over a billion users of YouTube and Zheng et al. [60] points to a similarly scaled volume of items in Google News. This tremendous amount of online content can be overwhelming for users, providing too many choices for a user to grasp an overview of all the content provided. Thus, personalized content recommendation improves user experience by narrowing the corpus of content for the user to efficiently decide on which content to consume next. Specifically, OTT television services such as ZDFMEDIATHEK¹ put much manual labor into curation of the contents in their media platform. Automation of this time-consuming task could free resources for more content production or improved quality assurance.

Contrary to many recommendation scenarios in other domains including online shopping and movie databases, news recommendation needs to rapidly adapt to a changing corpus of news to recommend, since news are outdated quickly. According to Zheng et al. [60], news are only of interest during the first 4.1 hours after publication. Furthermore, user preference in certain topics of news often changes over time [60]. For instance, a user may be interested in weather updates in the morning and a daily recap in the evening. Furthermore, preferences may change over larger periods of time, e.g., during different seasons or events such as soccer tournaments. These dynamics are impossible to handle manually and further prove the necessity of automation in news recommendation at ZDFMEDIATHEK.

Besides, personalized recommendations directly impacts economical development of a platform: Lamere and Green [25] reported that 35% of all sales on Amazon originated from recommended products and Das et al. [10] increased traffic on Google News by 38% by introducing a personalized recommender system. Similarly, ZDF as a publicly held company aims to increase usage of ZDFMEDIATHEK in order to defend public funding into its services and, possibly, advocate increased public funding in the future. Therefore, ZDFMEDIATHEK already shows a row of recommendations on its home page, as shown in Figure 1. Typically, these recommendations are ranked individually for each user.

Recommender systems take into account the content already consumed by a certain user in order to recommend them new items from a corpus of consumable items. Such systems solely relying on content, i.e., Content-Based (CB) approaches, often exhibit inferior performance compared to techniques utilizing both content similarities and user similarities, i.e., Collaborative

¹ ZDFmediathek: <https://zdf.de>

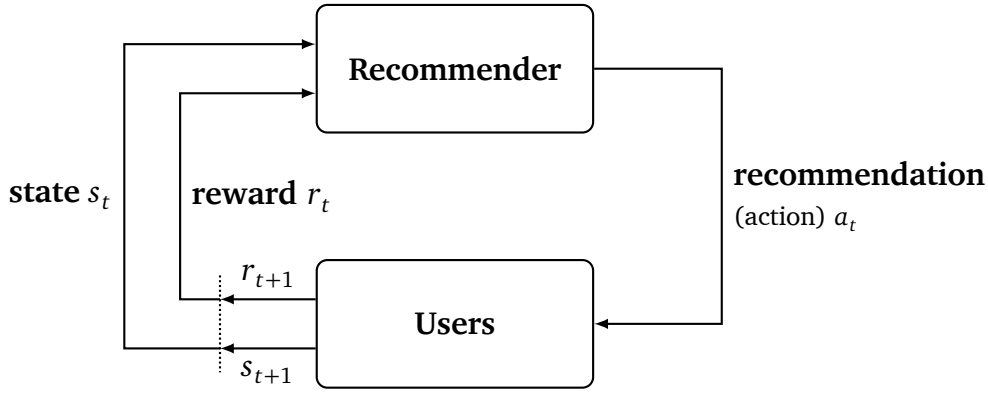


Figure 2: Reinforcement Learning in Recommender Systems

Filtering (CF) approaches. While improving on CB systems, CF approaches consider user preferences as static and, thus, are limited in their effectiveness in the dynamic nature of news recommendations. For instance, a user may be very interested in news about a particular thunderstorm approaching his location, which does not necessarily indicate a long-lasting user preference in thunderstorms. Besides, CB approaches require a dense item-to-item similarity matrix and CF techniques are based on user-item matrices. Data for these matrices is often sparse and, thus, most prior work does not handle the fast pace of news items appearing and becoming irrelevant. Furthermore, both CB and CF maximize immediate rewards such as click-through rates instead of taking into account long-term benefits such as user activeness on the platform or user return patterns. To counter these limitations, recent approaches in news recommendation utilize Deep Learning (DL) to learn more complex user-item interactions and apply Reinforcement Learning (RL) for considering expectations of future rewards in addition to immediate rewards [31; 60].

The aforementioned combination of improvements in user experience, increase in productivity and economic growth through recommendations and limitations of state-of-the-art approaches convinced ZDFMEDIATHEK to implement a DL based approach specifically for news recommendations. Zheng et al. [60] propose a novel approach using Deep Q-Learning to solve the online personalized news recommendation problem. News recommendation requires a system to handle dynamic changes instead of a static corpus of content, user activity instead of click labels and diversification instead of recommending similar items. Therefore, Zheng et al. [60] propose a Double Dueling Deep Q-Network (DDQN) framework to solve these challenges by continuously retraining through RL, explicitly considering future reward, maintaining a user activeness score and pursuing more exploration using a Dueling Bandit Gradient Descent (DBGD) method.

RL adequately handles a dynamically changing corpus of news and user preferences by continuously retraining the learning model. Figure 2 depicts the interaction of the recommender system with the users in a RL context. The recommender system provides recommendations to the users. Afterwards, the users provide feedback on the recommendations by either explicitly rating or implicitly clicking and consuming the recommended items. The recommender system receives this feedback as reward for the provided recommendations and, furthermore, receives an updated state with current user preferences and an extended corpus of news to recommend. Obviously, this system has a drawback: in the beginning, the recommender system needs to provide recommendations without “knowing” the state and reward. We handle this *cold-start problem* by pre-training the DDQN in an offline simulator based on recorded user interactions in ZDFMEDIATHEK.

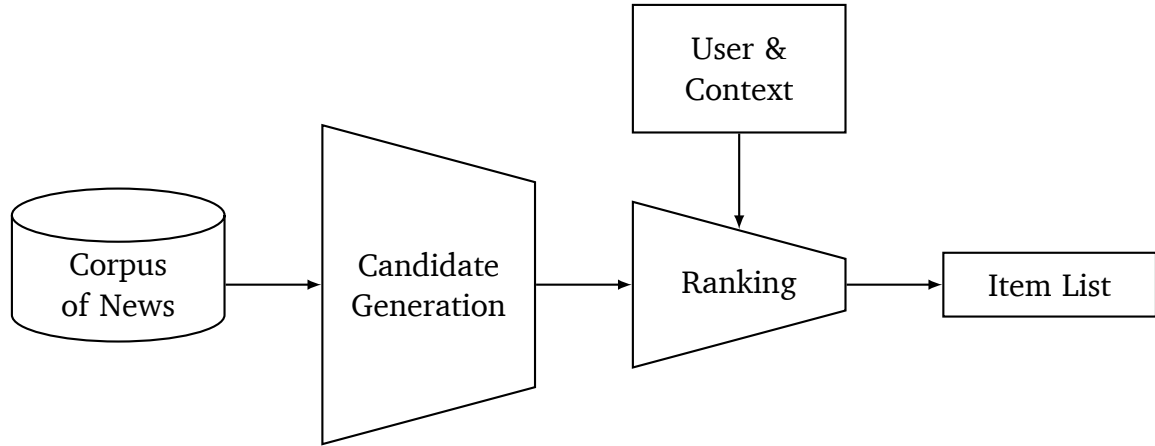


Figure 3: Candidate Generation in Recommender Systems [c.f., 9, p. 2]

The underlying Deep Q-Network (DQN) of Zheng et al. [60] utilizes Q-learning to decide on a ranking for recommended items. The DQN assigns a Q-value to each item in the corpus denoting the item’s probability of bearing a high reward, i.e., being clicked or increasing user activeness on the platform. Unfortunately, this requires each item in the corpus to be put into the DQN to predict a Q-value. Since this does not scale well for large corpora, we only calculate Q-values for a pre-determined set of candidates. Figure 3 shows the pipeline starting with the corpus of news. Next, the pipeline passes the candidate generation that reduces the amount of items to be considered in the ranking step. This ranking step produces a ranked list of recommended items considering the context and the requesting user’s preferences and history. In other domains apart from news recommendation, complex candidate generation methods may be necessary. But due to the very dynamic nature of news, which are outdated after only 4.1 hours, naïvely selecting the freshest news as candidates is adequate [60]. Furthermore, we represent states and actions in continuous spaces, allowing for consideration of previously unknown state and action features in the recommendation procedure. Thus, the DQN appropriately handles arbitrary candidates.

To sum up, we aim to recreate the DDQN approach by Zheng et al. [60] featuring two dueling Deep Q-Networks in a double Deep Q-Network architecture and an exploration DQN in Dueling Bandit Gradient Descent. Unfortunately, recreation of published papers’ results is often complex and challenging, although it assures scientific confirmability and interpretability [3]. Furthermore, we aim to adapt this approach to a similar domain, ZDFMEDIATHEK, which provides sufficient data and allows to test transferability of the DDQN agent to similar domains.

The following thesis is structured as follows: first, section 2 examines existing approaches and provides preliminaries for Reinforcement Learning, Deep Learning and the combination of both in Deep Q-Networks. Second, we present our model architecture of the DDQN adapted for ZDFMEDIATHEK in section 3. Next, we discuss implementation details and evaluation details in section 4 and section 5 respectively. Finally, we present closely related work in section 6 and come to a conclusion in section 7.



2 Background

Since news recommendations are a very narrow field of studies, we reference prior work on generating recommendations in general, narrowing down to Reinforcement Learning based approaches and, finally, name a few news recommendation techniques. Furthermore, we provide the theoretical fundamentals required for the Double Dueling Deep Q-Network utilized in our technique.

2.1 Prior Work

Recommender systems have been studied extensively, providing numerous existing approaches. These approaches include content-based filtering [39], matrix-factorization-based methods [11; 24; 30; 53], logistic regression [36], factorization machines [19; 40] and, recently, deep learning models [9; 31; 54; 60].

Content-based filtering recommends items by considering content similarity between items [39]. In contrast, Collaborative Filtering (CF) recommends items preferred by users with similar behaviour and assumes that similar users tend to provide the same ratings for items. Thus, conventional CF-based methods rely heavily on existing ratings to calculate similarities in order to provide reliable recommendations and, therefore, are prone to errors through scarce data. Recommender systems often make use of advanced CF-based methods such as Matrix Factorization (MF). MF represents both items and users as vectors in the same space [11; 24; 30; 53]. Besides, recommendation may be presented as binary classification problem, i.e., whether to recommend an item or not. Logistic Regression (LR) solves such binary decision problems. However, LR-based approaches are „hard to generalize to the feature interactions that never or rarely appear in the training data“ [c.f. 31, p. 3]. Conversely, Factorization Machines (FM) show promising results even on scarce data by modeling pairwise feature interactions as inner product of latent vectors corresponding to features. Recently, the complex feature interactions for recommendation procedures were learned by deep learning models [9; 31; 54; 60].

Contextual Multi-Armed Bandits for Recommendations

Additionally to previously mentioned approaches, *Contextual Multi-Armed Bandit (MAB) models* were utilized to generate recommendations [7; 27; 52; 57; 59]. MABs are a group of recommender systems, that select a different *arm*, i.e., recommendation technique, for each request based on the probability of this arm’s recommendations yielding a high reward. In Contextual MABs depicted in Figure 4a, the *bandit* considers the *context*, i.e., certain features concerning the bandit’s task, when selecting one of its arms. For news recommendations, this context contains both user and item features. Zeng et al. [57] already considers dynamic user preferences varying over time.

However, all previously named approaches, including CF, MF, LR, FM, and, Contextual MABs, face two limitations regarding news recommendations.

1. These approaches consider the user’s preference as static and aim to learn this preference as precise as possible.

In news recommendation, user interest is especially volatile and even changes in the course of a single day [60]. Thus, the recommendation procedure cannot be modeled as static process.

2. Furthermore, all aforementioned techniques aim to maximize immediate rewards, e.g., user clicks.

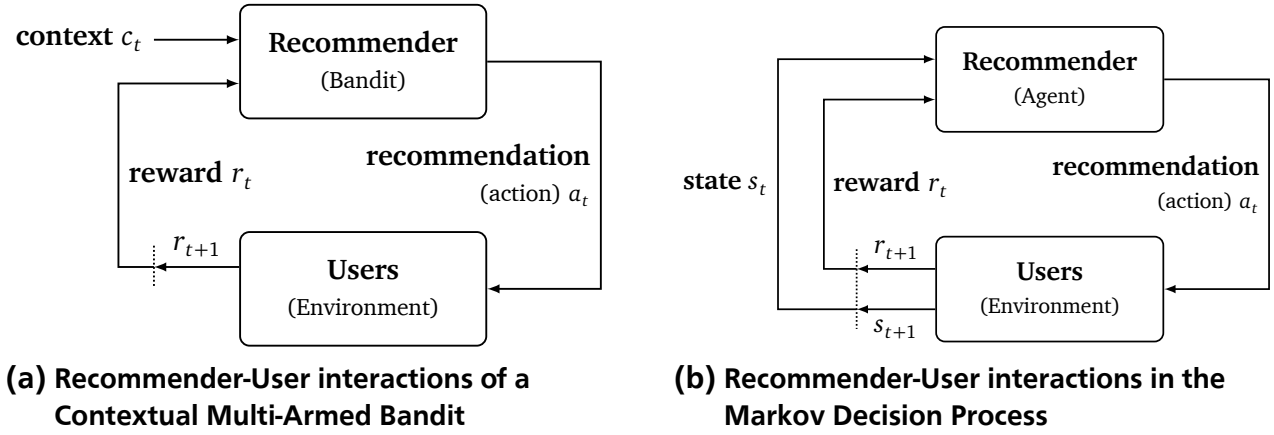


Figure 4: Comparison of Contextual Multi-Armed Bandit and Markov Decision Process

Only considering immediate rewards could harm recommendation performance in the long term. In addition to immediate rewards, we aim to account for long-term benefits of recommendations such as increasing user activeness on the platform under consideration.

Reinforcement Learning for Recommendations

Furthermore, Reinforcement Learning (RL) was applied to recommendation scenarios as *Markov Decision Process (MDP) models*. In the MDP, an *agent* selects *actions* based on his perception of the *environment* to improve the agent’s position in the environment. The agent’s position is calculated based on *reward* received from the environment after presenting the selected actions. As shown in Figure 4b, a recommender system acts as the agent in RL recommendations. The agent’s action space is represented by the item space from which the recommender system selects the best items to present to the users, who act as an MDP environment. Based on the recommended items, users generate rewards for the recommender system, e.g., by clicking recommended items. Next, the recommender system receives this reward and, in contrast to the previously described Contextual MAB, also perceives the new state created through the recommender’s user interaction. The recommender system continuously adapts its recommendation policy to generate recommendations yielding a higher reward.

Contrary to Contextual MABs, agents in an MDP are capable of considering potential future rewards [60]. Many previous approaches try to model the items as state and the transition between items as action, leading to an exploding state space for larger corpora of items [32; 34; 41; 45; 49]. Furthermore, training these models is limited by sparse transitions data [31; 60]. In contrast to prior work, we design continuous state and action spaces, which allows for scaling of the corpus of news and is robust to sparse interaction data.

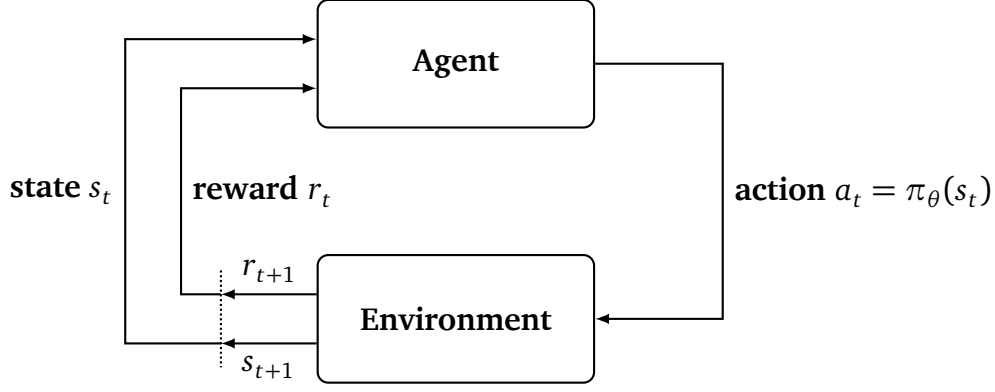


Figure 5: Agent and Environment in the Markov Decision Process [31]

2.2 Preliminaries

Since our technique combines multiple machine learning paradigms, such as a Q-learning approach to Reinforcement Learning and Artificial Neural Networks in a Deep Learning manner, we first introduce the theoretical fundamentals of these paradigms. Next, we describe how these paradigms are combined in a Deep Q-Network for Deep Reinforcement Learning.

2.2.1 Reinforcement Learning

An *agent* in a Reinforcement Learning (RL) scenario interacts with its environment over time, thus, the agent reinforces its behaviour by continuously learning new information based on its (inter-)actions and the environment's reaction. A Reinforcement Learning agent is formalized in the Markov Decision Process (MDP). We define an MDP as $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ with

- \mathcal{S} denoting the *state space*,
- \mathcal{A} denoting the *action space*,
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$ denoting the *state transition function*,
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ denoting the *reward function*, and,
- γ denoting the *discount rate*

of the agent [1; 28; 29; 31; 48]. Figure 5 visualizes an agent in an MDP: At each time step t , the agent receives a state $s_t \in \mathcal{S}$ and selects an action $a_t \in \mathcal{A}$ following a policy $\pi_\theta(a_t|s_t) : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ describing the agent's behavior. In the next time step, the agent receives state $s_{t+1} \in \mathcal{S}$, following the agents behavior (selection of a) and environmental dynamics modeled in $\mathcal{P}(s_t, a_t, s_{t+1})$. Finally, the agent receives a reward r_t according to the reward function $\mathcal{R}(s_t, a_t, s_{t+1})$.

Temporal Difference Learning

Generally, an agent in an MDP aims to find an *optimal policy* $(\pi^* : \mathcal{S} \times \mathcal{A} \mapsto [0, 1])$ which maximizes the *expected cumulative rewards* from any state $s \in \mathcal{S}$, i.e.,

$$V^*(s) = \max_{\pi_\theta} \mathbb{E}_{\pi_\theta} \left\{ \sum_{t=0}^{\infty} \gamma^t r_t \right\}. \quad (1)$$

Here, $V^*(s)$ is the value of state s under the optimal policy, \mathbb{E}_{π_θ} is the expectation under policy π_θ , t is the current time step and r_t is the reward at time step t discounted by the discount rate γ . Discounting by γ^t values immediate rewards higher than expected future rewards to account for uncertainty of the future.

A central aspect of RL is *temporal difference learning*, which allows to learn the value function $V(s)$ directly from the temporal difference error $V(s_{t+1}) - V(s_t)$ through *value iteration* of the *Bellman approximation*

$$V_s \leftarrow r + \gamma \max_{a' \in A} V_{s'} \quad (2)$$

for each tuple of state s , action a , reward r for the action and next state s' [47]. The Bellman approximation in Equation 2 approximates the Bellman equation of optimality $V_0 \leftarrow \max_{a \in 1 \dots N} (r_a + \gamma V_a)$ which Richard Bellman proved to always find the optimal policy [2]. For smoother convergence of V_s , we blend old and new values

$$V_s \leftarrow (1 - \alpha)V_s + \alpha(r + \gamma \max_{a' \in A} V_{s'}) \quad (3)$$

given a constant step-size parameter $\alpha \in (0, 1]$, also known as *learning rate* in the machine learning context [26; 28; 29; 48, pp. 25, 97].

Q-Learning

Equivalently to maximizing $V(s)$ in temporal difference learning, an agent in an MDP may maximize the expected cumulative rewards from any state-action pair $\{(s, a) | s \in \mathcal{S}, a \in \mathcal{A}\}$, i.e.,

$$Q^*(s, a) = \max_{\pi_\theta} \mathbb{E}_{\pi_\theta} \left\{ \sum_{t=0}^{\infty} \gamma^t r_t \right\}. \quad (4)$$

Here, $Q^*(s, a)$ is the value of taking action a in state s under the optimal policy, \mathbb{E}_{π_θ} is the expectation under policy π_θ , t is the current time step and r_t is the immediate reward at time step t discounted by the discount rate γ . Again, discounting is applied to account for uncertainty of the future.

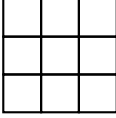
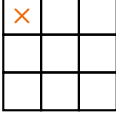
Similarly to value iteration, we can directly learn $Q(s, a)$ from the temporal difference error in a process named *Q-Learning* through the Bellman approximation

$$Q_{s,a} \leftarrow r + \gamma \max_{a' \in A} Q_{s',a'} \quad (5)$$

since $V^*(s) = \max_a Q^*(s, a)$ holds [48, pp. 51, 107; 56]. Analogous to value iteration, we blend old and new values of $Q_{s,a}$ for smoother convergence

$$Q_{s,a} \leftarrow (1 - \alpha)Q_{s,a} + \alpha(r + \gamma \max_{a' \in A} Q_{s',a'}). \quad (6)$$

The basic form of Q-learning is tabular Q-learning, which describes learning a state-action table of Q-values, where each cell contains the Q-value for the respective action in the corresponding state. Thus, the table includes return values of $Q(s, a)$ for all combinations of s and a . For instance, in a game of *Tic-Tac-Toe* each row of the table corresponds to one state of the board and each column of the table represents putting a marker on one of the fields. Tabular Q-learning starts with an empty table in state s_0 and repeatedly performs the Bellman update as described in Equation 6 to fill the cells [26, p. 121]. After multiple iterations of Q-learning, the agent's Q-table may contain similar Q-values to the following table:

Q-Table of Tic-Tac-Toe						
State	Top Left	Top Center	...	Center Center	...	Bottom Right
$s_0 =$ 	0.8	0.4	...	0.1	...	0.7
$s_1 =$ 	0.0	0.1	...	0.9	...	0.2
...

Here, bold Q-values represent the best action in each state, i.e. $\max_a Q(s_t, a)$. Thus, as shown in the first row, the agent would start the game by putting its marker in the top left cell of the board, i.e., $\operatorname{argmax}_a Q(s_0, a)$. If the agent needed to react to the out-coming state s_1 as shown in the second row, the agent would put its marker into the center of the board. We omitted all further states and actions for comprehensibility.

Naturally, learning a Q-table works well for relatively small and static state and action spaces such as Tic-Tac-Toe. For large state and action spaces as in recommender systems, the Q-table approach is by far too memory-intensive. Furthermore, Q-tables cannot handle dynamic state and action spaces and struggle with sparse information. Thus, recommendations are only possible with a more efficient and robust approach, e.g., by approximating the Q-function with an Artificial Neural Network.

Sutton and Barto [48] provide proves and an in-depth explanation on Reinforcement Learning beyond the scope of this thesis. Additionally, Géron [12] and Lapan [26] provide insight about practical implications of RL, temporal difference learning and Q-learning.

2.2.2 Neural Networks & Deep Learning

As Deep Learning (DL) is a specialized technique utilizing Artificial Neural Networks (ANNs), we first describe the foundations of neural networks. An ANN consists of a multitude of artificial *neurons* which are connected through *synapses* as shown in Figure 6, following the nomenclature of a human brain [21]. Each neuron has one or more inputs, an activation threshold and at least one output that sends a signal via a synapse to another neuron upon activation. The connection of multiple neurons and synapses is called *perceptron*. Usually, perceptrons are organized into multiple interconnected layers in a Multilayer Perceptron (MLP), the base architecture of a trivial ANN.

Formally, a neuron has multiple inputs x_1, x_2, \dots, x_n weighted with corresponding weights w_1, w_2, \dots, w_n , as depicted on the left side of Figure 6. Additionally, each neuron has a bias, represented by $(+1) \times w_0$. The neuron calculates the sum of the bias and all weighted inputs, i.e., $w_0 + \sum_{i=1}^n w_i x_i$. This sum is used as input of the neuron's *activation function* σ . Here, σ acts as an activation threshold that controls whether the neuron is activated and “fires” a signal to all dependent neurons.

The right side of Figure 6 shows the interdependence of neurons. The neurons are organized into *layers*. Here, the inputs I_1, I_2, I_3 each reach one neuron in the *input layer*. Each neuron in

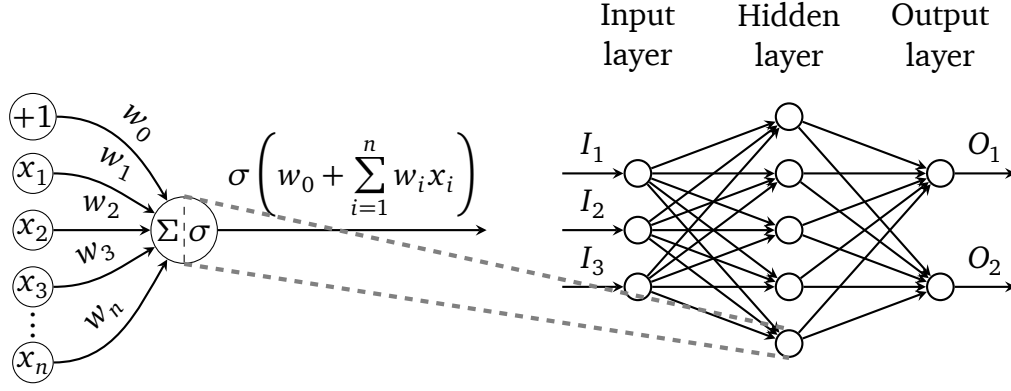


Figure 6: Artificial Neuron in a Multilayer Perceptron [51]

the input layer then applies its activation function and feeds the output to all neurons in the so-called *hidden layer*, which is not “visible” from the outside because no inputs or outputs are connected to its neurons. Each neuron in the hidden layer applies its activation function to its inputs and feeds the result to both neurons in the *output layer*. Finally, the neurons in the output layer produce the outputs O_1, O_2 by applying their activation functions. Naturally, this model is called *Multilayer Perceptron* since it is a perceptron consisting of multiple layers of neurons, or *Feed-Forward Neural Network*, since all synapses feed-forward to the following layer instead of allowing recurrent connections.

The number of neurons per layer determines the *width* of this layer, while the number of layers determines the *depth* of the neural network. Deep Learning utilizes “deep” neural networks, thus, ANNs with multiple hidden layers [14].

Such ANN is usually trained through a process called *backpropagation* [44, p. 1]. Basically, backpropagation trains an ANN by adjusting the weights of the synapses between neurons „in proportion to the product of their simultaneous activation“ [c.f. 35, p. 1], thus, „propagating corrections back towards the sensory end [i.e., input layer] of the network if it fails to make a satisfactory correction quickly at the response end [i.e., output layer]“ [43, p. 292]. Since backpropagation adjusts the weights through gradient descent on the “propagated corrections“, all activation functions in the ANN need to be differentiable.

First, the network is fed-forward from the input layer to the output layer with a training example to infer a prediction. During the feed-forward step, each neuron stores its output and partial derivatives of its activation function for each input. At the output layer, a loss function E is applied to the predicted output and the expected output, calculating the error. Afterwards, this error is backpropagated iteratively through the network: for each weight w_{ij} at the synapse from neuron i with output o_i to neuron j the gradient of the loss function is calculated as

$$\frac{\partial E}{\partial w_{ij}} = o_i \frac{\partial E}{\partial o_i w_{ij}} = o_i \delta_j \quad (7)$$

where δ_j denotes the *backpropagated error* up to neuron j . Once all partial derivatives of E have been computed, the weights w_{ij} are updated via gradient descent:

$$w_{ij}^* = w_{ij} - \gamma o_i \delta_j \quad (8)$$

with *learning rate* γ to only train the network *towards* the current training example but keep experience from previous examples.

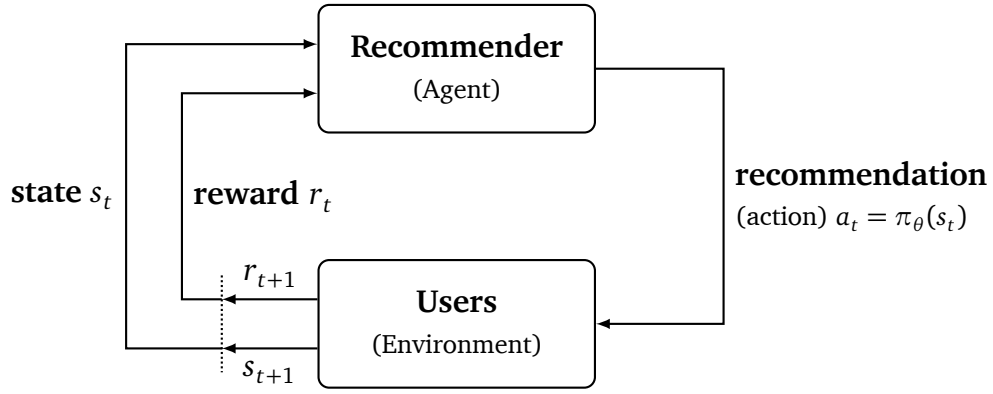


Figure 7: Recommender-User Interactions in the Markov Decision Process [31]

Rojas [42] more thoroughly describes and proves the backpropagation algorithm in more complex network architectures and various forms of model formalization, i.e., in matrix form.

2.2.3 Deep Reinforcement Learning

Deep Reinforcement Learning describes approaches utilizing Deep Learning to train an agent in a Reinforcement Learning context, which has already proven to be beneficial in music recommendation [54] and YouTube video recommendation [9].

Often, Deep Learning techniques are utilized to approximate the Q-value described in subsection 2.2.1 in an approach named Deep Q-Network (DQN) [38]. To achieve this, an ANN is used for the agent’s implementation. Since ANNs require vast amounts of training data, the agent may be pre-trained on historical data of interactions from a similar agent with the environment before starting the actual RL process to refine the agent over time. Afterwards, this ANN is an approximator for the Q-function under the optimal policy $Q^*(s, a)$ (c.f., Equation 4).

In a recommendation scenario, the RL terminology may be replaced by the corresponding terms from recommender engines. As shown in Figure 7, we map the recommendation procedure onto a sequential decision making problem where the recommender acts as an agent, providing recommendations instead of actions to users, who act as an Environment by providing feedback on the recommendations via clicks, ratings or consumption times. The recommender uses the quantified user feedback as reward for past recommendations to improve in the future.



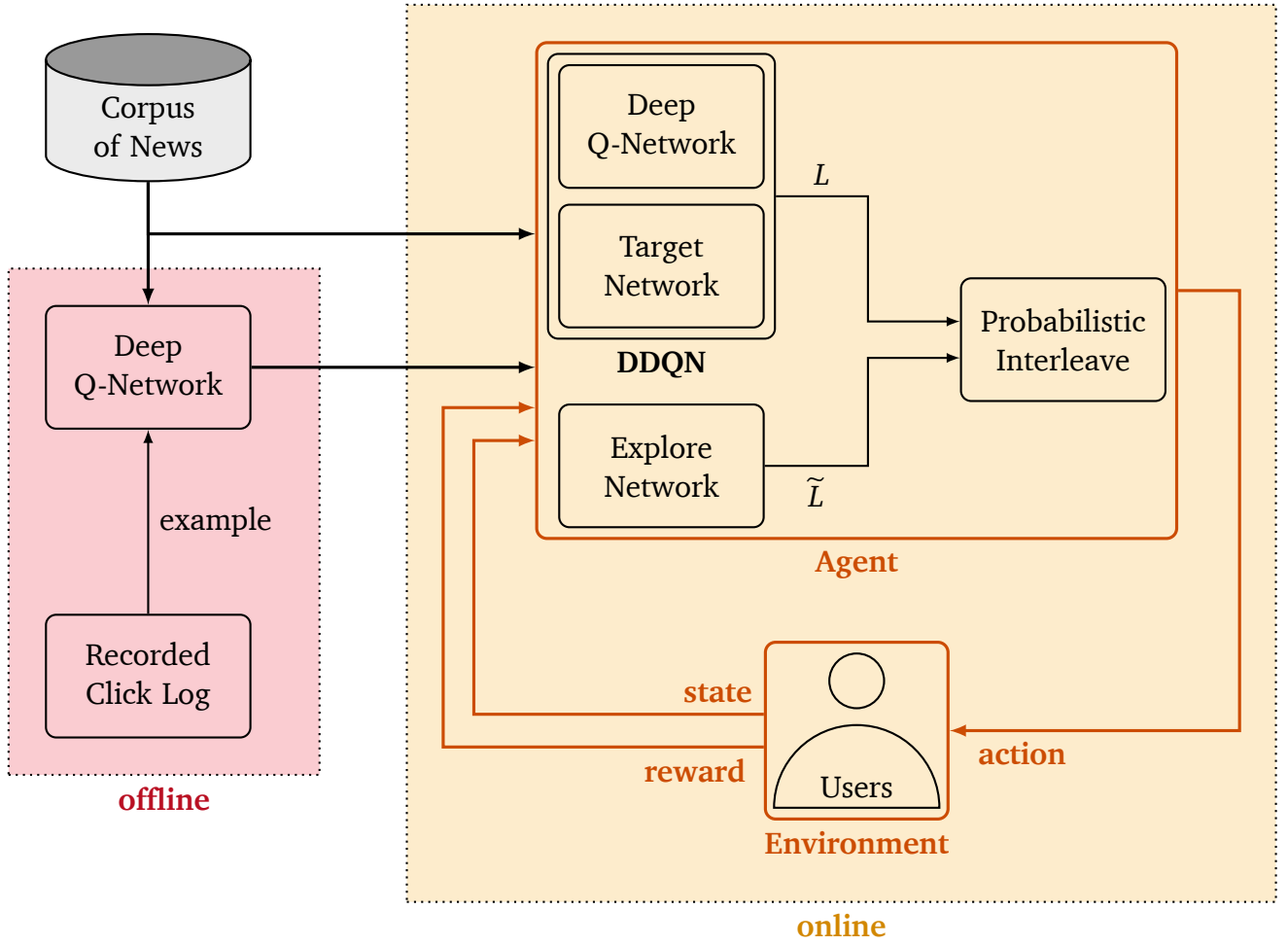


Figure 8: Conceptual View on the Technique in MDP

3 Technique: DDQN

Previous methods lack key features for news recommendation, especially regarding adaptability to a dynamically changing corpus of content, consideration of user activity and activeness instead of click labels or ratings and diversification to also present different, up-to-date news and prevent formation of a filter bubble [60]. Therefore, we examine a novel approach by Zheng et al. [60] specifically designed for news recommendation and adapt the approach to data and requirements found in a commercial real-world scenario: ZDFMEDIATHEK². The news domain offers a specifically large corpus of items for recommendation. Furthermore, this corpus of news items is very dynamic, requiring continuous updates of the recommendation engine. Besides, user preferences change quickly based on context, i.e., time of day, day of week or season. For instance, a user may only be interested in soccer during international tournaments, requiring the recommender engine to explicitly consider user preference in the recommendation procedure.

Figure 8 presents a conceptual view on our technique. Naturally, we depend on the corpus of news shown in the top left. Besides, we clearly differentiate between an offline part on the left and an online part on the right. Here, we apply machine learning terminology, where “offline” learning has access to all data from the beginning in contrast to “online” learning, where data becomes available sequentially during the training process.

² ZDFmediathek: <https://zdf.de>

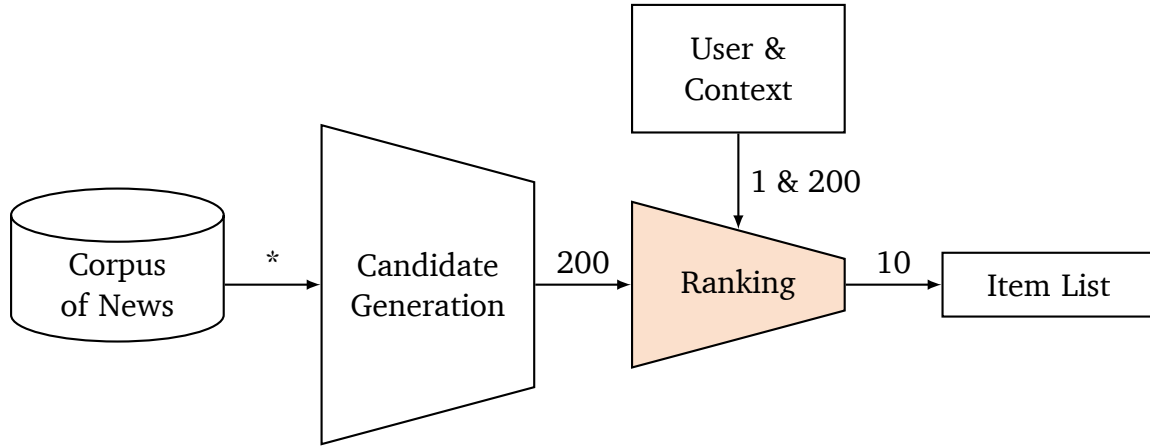


Figure 9: Candidate Generation in ZDFMEDIATHEK [c.f., 9, p. 2]

In offline training, we train the Deep Q-Network (DQN) on examples extracted from a click log of previously recorded user interactions with ZDFMEDIATHEK. Then, we deploy the trained DQN into real-world recommendation and online learning in ZDFMEDIATHEK. This section covers both offline and online training of the DQN and describes the architecture of our agent in Markov Decision Process (MDP).

3.1 Recommendation Pipeline

In ZDFMEDIATHEK we have access to an enormous amount of news to recommend to the user. If we were to input all of the news into a neural network, we would immediately reach memory and calculation limits of state-of-the-art hardware. Thus, we optimize the recommendation pipeline by injecting a *candidate generation* before actually considering any news for recommendation. As depicted in Figure 9, the candidate generation step acts as a funnel in the process, dramatically reducing the size of considered news to 200 candidates. Here, we select the newest 200 news from the corpus, because news are outdated after only 4.1 hours [60]. The amount of news selected here is arbitrary, although it massively impacts model size. The model size is constraint by memory capacity and performance limits, thus, a sufficiently small number of candidates is required.

Our technique covers the following *ranking* step, which ranks the provided candidates under consideration of provided user and context features to produce a list of the 10 best items. Again, the decision for 10 items is arbitrary. Independently of the length of the output list, we learn Q-values for all 200 provided items and, thus, only filter and rank the top 10 items afterwards. This cutoff represents a threshold to divide between relevant items presented to the user and irrelevant items held back.

To sum up, our technique performs only the ranking step in Figure 9 on sufficiently small candidate sets to adhere to memory constraints. We consider candidate generation as given, which could be optimized in future work.

3.2 Model Architecture

The base architecture for our model is a Deep Q-Network (DQN). Figure 10a conceptually shows a basic Deep Q-Network, which only consists of fully-connected sequential layers. Thus, all layers of the DQN are combined to form a function approximator for $Q(s, a)$. Directly approximating $Q(s, a)$ often yields overoptimistic values for estimated Q-values [55].

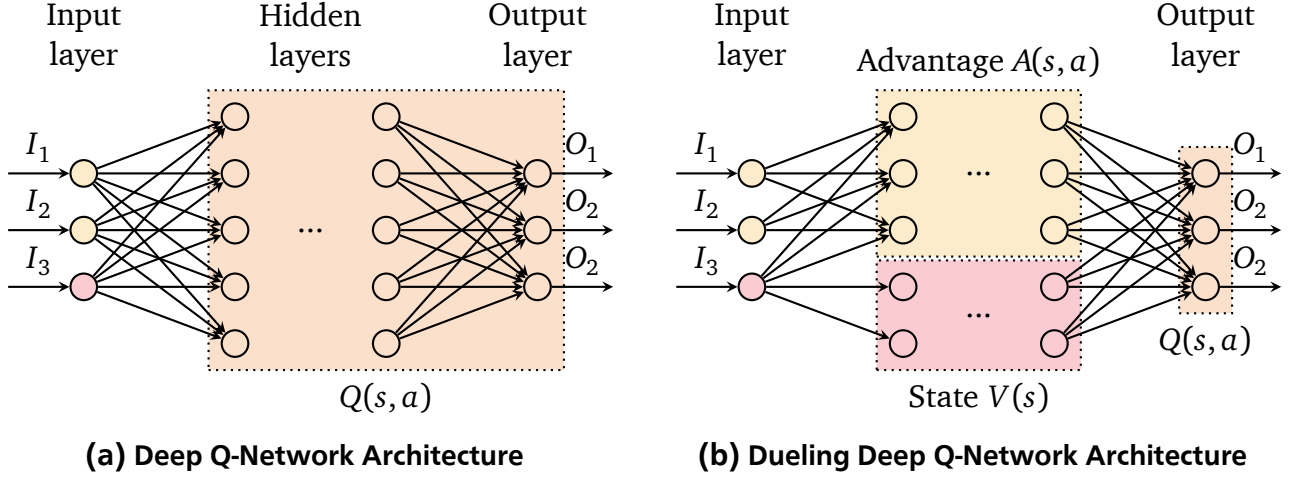


Figure 10: Comparison of Network Architectures [based on 51]

For improved performance, we use the advanced dueling Deep Q-Network (dueling DQN) architecture, which splits value function $V(s)$ and advantage estimation $A(s, a)$ into dueling branches within the model, as shown in Figure 10b [55]. The branch for advantage estimation $A(s, a)$ in the center top of Figure 10b receives both state features s and action features a . In contrast, the branch for value estimation $V(s)$ in the bottom center of Figure 10b receives only those features describing state s . In the output, both advantage prediction and value approximation are recombined into Q-value estimations $Q(s, a) = V(s) + A(s, a)$.

Here, the advantage $A(s_t, a)$ denotes only the estimated delta between $V(s_t)$ and $V(s_{t+1})$. Thus, $V(s_{t+1}) = V(s_t) + A(s_t, a)$ holds iff the prediction of $A(s_t, a)$ is correct, i.e., for the optimal advantage estimation and value function $V^*(s_{t+1}) = V^*(s) + A^*(s_t, a)$. As shown in subsubsection 2.2.1, $V^*(s) = \max_a Q^*(s, a)$ holds for the optimal Q-estimation and value function, implying that

$$\max_a Q^*(s_{t+1}, a) = V^*(s_{t+1}) = V^*(s_t) + A^*(s_{t+1}, a) \quad (9)$$

holds. Therefore, we are allowed to split the Q-estimation into value function and advantage estimation to improve prediction performance over the DQN [26, p. 191; 55].

Furthermore, our approach features two DQNs, i.e., a double Deep Q-Network (double DQN). Both DQNs in the double DQN share the same architecture, independently of the underlying DQN architecture, e.g. plain or dueling DQN. One of the networks in the double DQN, the *q-network*, acts as conventional DQN, i.e., selecting the highest-rated items based on its predicted Q-values. In contrast to single-network DQNs, the second network of a double DQN, the *target network*, rates the selected actions of the first network. Based on the target network's ratings, we adapt the Q-learning Bellman update introduced in section 2 from

$$\begin{aligned} Q_{s,a} &\leftarrow r + \gamma \max_{a' \in A} Q_{s',a'} \quad \text{c.f., Equation 5} \\ &\Rightarrow Q(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) \end{aligned} \quad (10)$$

to the following form by incorporating Q-estimations \tilde{Q} from the target network:

$$Q(s_t, a_t) = r_t + \gamma \max_a \tilde{Q}(s_{t+1}, \arg\max_a Q(s_{t+1}, a)) \quad (11)$$

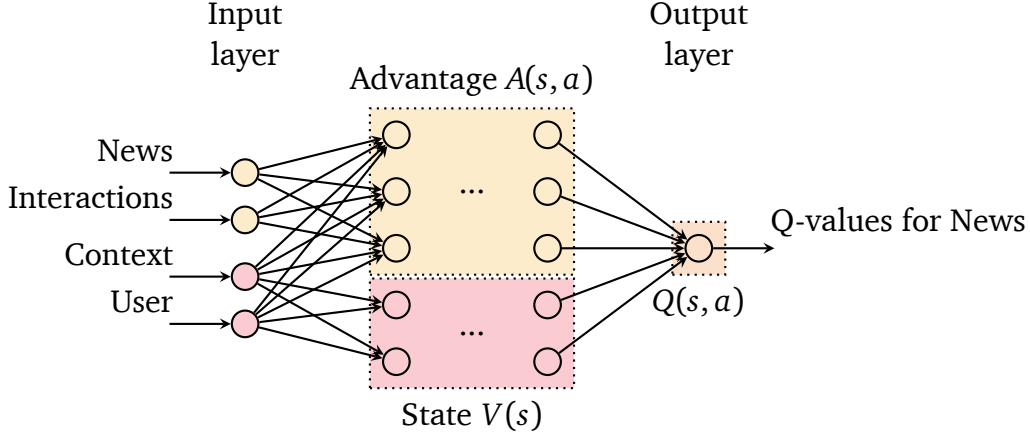


Figure 11: Dueling Deep Q-Network Architecture in ZDFMEDIATHEK

where $\operatorname{argmax}_a Q(s_{t+1}, a)$ denotes the best action selected by the Q-network, i.e., the highest-rated action a in state s_{t+1} . In $\max_a \tilde{Q}(s_{t+1}, \dots)$ the target network re-evaluates action a from the Q-network by assigning its own Q-value to the action selected by the Q-network. The target network's Q-value is then used for the Bellman update of $Q(s_t, a_t)$ by discounting it with γ and adding the reward r_t . Using this adapted Bellman update with double DQN effectively counters overestimation tendencies of DQN approaches [50]. As described before, we improve robustness of the Bellman update by blending old and updated values together via a learning rate α :

$$Q(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_a \tilde{Q}(s_{t+1}, \operatorname{argmax}_a Q(s_{t+1}, a))) \quad (12)$$

Our Double Dueling Deep Q-Network (DDQN) combines the double DQN approach with a dueling DQN architecture. Thus, we have both Q-network and target network in a double DQN, which share the same dueling DQN architecture with differing weights.

3.3 Input Features

As shown in Figure 11 we utilize four different kinds of features to compute Q-values for all provided news features. The four kinds of features are mapped onto the input layers of a DDQN. These features include user features, news features, context features, and interaction features.

News features describe news entries in ZDFMEDIATHEK in a mostly one-hot encoded form, including publication and editorial dates (in seconds since January 1st, 1970), video metadata, visibility information, and one-hot encoded brand ids and news types. In total, these features are 367-dimensional.

Context features describe the context of a news request, i.e., the date and time when the request happens and the freshness of accompanying news features at that time, i.e., time delta from publish date until request time. Both these times are given as absolute timestamps in seconds since January 1st, 1970 and, additionally, the freshness is provided in relative years, months, days, and hours in 38-dimensional feature vectors.

Interaction features describe past interactions of users with news items. These features consist of the amount of viewing minutes, a coverage score, and 18 one-hot encoded genres. Here, each 20-dimensional feature describes one user-news interaction, i.e., one click on a news item or one play event of a video in ZDFMEDIATHEK.

User features describe the interests of users in a mostly one-hot encoded form analogous to the format of news entries. A user is represented as the aggregated interactions within the past hour, past 6 hours, past day, past week, and past year relative to a given request time. These five aggregations are concatenated, giving a 1920-dimensional feature vector, which equals five times the size of news features without video duration and user-news features without viewing minutes and coverage.

Each request to the DDQN recommendation engine requires one user feature, for whom the requests are provided, several news features as candidates to draw the recommendations from, a context feature for each provided news candidate and multiple user-news features for consideration of the user’s recent interactions.

Figure 12 shows an actual summary of the network from a trained Tensorflow model with exact dimensions of each layer given as tuple. The first value of a dimension tuple denotes the batch size, which is always *None* until instantiation of the model. The last value of a tuple is the feature length, i.e., the length of a single feature vector. If the tuple contains three values, the center specifies the cardinality of feature vectors provided per request. Here, we provide 200 *news* and *context* features per request with 367 and 38 dimensions and twenty 20-dimensional *interaction* features, but only one 1920-dimensional *user* feature per request.

One-Hot Encoding

The input features are mostly one-hot encoded, because neural networks only consider floating point literals and, thus, cannot work with categorical data. For example, the categorical input “brand”, which specifies the specific studio that created a news item, is provided as alphanumeric id (*heute-19-uhr-102*) and name string (*heute 19:00 Uhr*) in the original input data. To use such categorical data as input to a neural network, we have to encode it into floating point literals or tensors.

For instance, we may assign an integer to each brand and store this mapping $brand_i \rightarrow i$ in a process called *label encoding*. This would create legal inputs to our DDQN, but the ordering within the mapping could influence prediction performance. Effectively, we would bias the network to consider brands mapped to 1 and 2 to be “more similar” than brands mapped to 1 and 5 [12].

Therefore, we encode the brands with *one-hot encoding*. Here, we assign a mapping $brand_i \rightarrow (a_0, \dots, a_n), a_i = 1, a_j = 0 : j \neq i$. The created vector (a_0, \dots, a_n) of size $1 \times n$ for n brands contains only one “hot” 1 at index i and $n - 1$ “cold” zeros at all other indices. Thus, all brands are orthogonal to one another in n -dimensional space, i.e., $(a_0, \dots, a_n) \cdot (b_0, \dots, b_n) = 0$.

However, one-hot encoding introduces two new disadvantages into our system. First, one-hot encoded features are vastly larger than label-encoded features. While label-encoding only requires a scalar for n categories, one-hot encoding requires a $1 \times n$ vector. Furthermore, one-hot encoding is static after initial mapping, i.e., we cannot expand an established one-hot encoding to more categories. For instance, we only consider a certain set of known “brands” for our one-hot encoding. If ZDFMEDIATHEK decides to add an additional brand to this set, we have to completely retrain the model with a larger vector for brands or set the brand to an all-zero vector for all newly-added brands. Nonetheless, the advantage of orthogonality between categories outweighs resource concerns. Moreover, categorical features such as brand are mostly static at ZDFMEDIATHEK.

3.4 Network Layers

Each of the four *Input* layers (user, interactions, context, news) is directly connected to a single *Flatten* layer. *Flatten* transforms $n \times m$ matrices into $1 \times (nm)$ vectors by concatenating all rows into a single row. For instance, all 200 news features of 367 dimensions are transformed into a single 73400-dimensional vector.

After flattening, all features share the same batch size and cardinality, i.e., only one row. This allows for concatenation of the features in the third layer of the DDQN as displayed in Figure 12. Here, we divide the features into *state features* and *action features*. State features represent the current state s , while action features are only relevant for action a . Since we split value calculation $V(s)$ and advantage estimation $A(s, a)$ in a dueling DQN architecture, we build one layer combining only the state features and one layer combining both state and action features. Layer `dueling_state_features` concatenates only the flattened versions of the state features *user* and *context*. In contrast, `advantage_features` concatenates both flattened state and flattened action features, i.e., *user*, *context*, *interactions*, and, *news*.

The following layers are split into dueling branches of $V(s)$ and $A(s, a)$, which we describe separately below. The branches are then merged into the final Q-value prediction layer as explained below.

Advantage Prediction

The advantage prediction $A(s, a)$ utilizes a simple sequential model of 3 layers. The first of these layers is the Dense layer `hidden_advantage_prediction`, i.e., all inputs in this layer are fully connected to all outputs. This layer reduces the immense 83320-dimensional combination of action and state features from the `advantage_features` layer into 512 neurons.

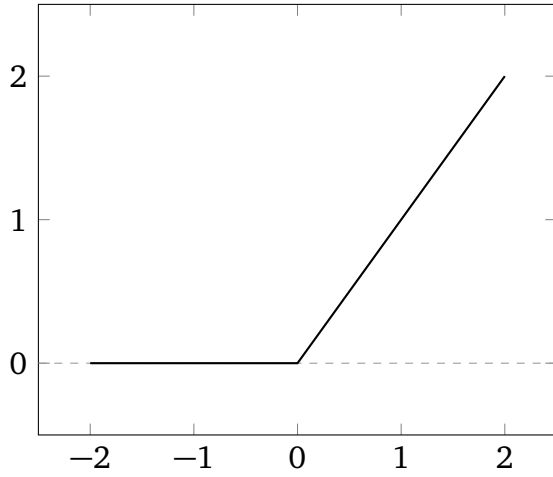
The next layer, `lrelu_advantage_prediction`, keeps these 512 dimensions, but uses a special *LeakyReLU* activation function instead of the default activation $wx + b$ with layer weights w , input x and bias b as defined in section 2. In contrast, the LeakyReLU (also known as LReLU) given in Equation 13 especially improves performance in deeper networks and is more robust to *vanishing gradients* than ReLU [33].

$$\text{LReLU}(x) = \max(w^T x, 0) = \begin{cases} w^T x & w^T x > 0 \\ 0.01w^T x & \text{else} \end{cases} \quad (13)$$

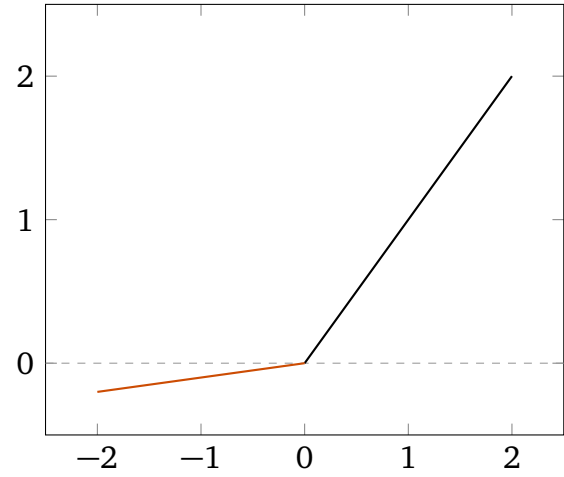
ReLU (*Rectified Linear Unit*), as defined in Equation 14, outputs 0 in the second case. This completely deactivates a neuron and may lead to it never being learnt through gradient descent in backpropagation. Hence, the neuron effectively *vanishes* from the network, naming this the *vanishing gradient problem*. LeakyReLUs improve upon this weakness by always activating marginally instead of outputting 0. Thus, LeakyReLUs are learnt in gradient descent, even if they are “inactive”.

$$\text{ReLU}(x) = \max(w^T x, 0) = \begin{cases} w^T x & w^T x > 0 \\ 0 & \text{else} \end{cases} \quad (14)$$

Figure 13 illustrates the difference between ReLU and LReLU activations. For demonstration intelligibility, Figure 13b has a larger “leak” of 0.1 instead of 0.01 in the second case of Equation 13.



(a) Rectified Linear Unit (ReLU)



(b) Leaky Rectified Linear Unit (LReLU)

Figure 13: Comparison of ReLU Activations

Following the LeakyReLU layer, another Dense layer reduces the 512 inputs to 200 outputs. Here, 200 denotes the number of Q-values the network should predict. This is equal to the number of provided news candidates in the news input layer. In a normal DQN, this layer could be the final output layer, optionally followed by a normalizing Layer such as Softmax. In contrast, our dueling DQN features a dueling value prediction branch.

Value Prediction

The value prediction $V(s)$ only uses state features provided by the `dueling_state_features` layer. First, we reduce these 9520 dimensions to 512 dimensions in the `dueling_hidden_state` layer. Next, we further reduce the 512 dimensions to a single value in `dueling_value_prediction`, i.e., we directly calculate $V(s)$ here.

Q-Value Prediction

Afterwards, both branches for value prediction $V(s)$ and advantage prediction $A(s, a)$ are combined through the following Lambda layer:

```
q_value_prediction = Lambda(lambda x: x[0] - mean(x[0]) + x[1],
                             output_shape=(num_actions,),
                             name="q_value_prediction")
([q_value_prediction,
  dueling_value_prediction])
```

This Lambda layer applies the lambda function given in its first parameter list to the layers given in its second parameter list. The lambda receives the prediction results of the layers `[q_value_prediction, dueling_value_prediction]` as parameter `x`, hence, `x[0] - mean(x[0])` calculates an averaged value of the Q-value $Q(s, a)$ for each news item provided and `x[1]` adds the value $V(s)$. This calculation results in an *output shape* of the Lambda layer of `num_actions= 200` which is the total amount of Q-values that should be computed.

3.5 Loss Function

The loss function determines, which metric is minimized during training of the network. Thus, the loss function defines the objective of the trained network. Therefore, the loss function needs to fit the problem definition.

Furthermore, since the DQN trains via gradient descent of the loss function, the loss depends on the state of the Q-network's weights. Since these weights are unknown before training, we initialize the network with randomized weights. Usually, this results in bad predictions, i.e., recommendations, and a high loss. Although reinforcement learning enables training the DQN in its real-world environment, we aim to maximize user satisfaction in ZDFMEDIATHEK. To counter this *cold-start problem*, we insert offline pre-training before deploying the DDQN agent to ZDFMEDIATHEK. In offline pre-training, our agent only replays recorded click logs, thus, only one item is clicked at a time and the agent cannot influence user decisions. Thus, we present two different loss functions for offline pre-training before deployment and online training of the agent in production, fitting the respective use cases.

3.5.1 Categorical Cross-entropy Loss

Tensorflow's `categorical_crossentropy` consists of two steps: It adds a Softmax activation to the network and applies the cross entropy cost function to the Softmax result.

First, categorical cross-entropy applies the Softmax function to the 200 Q-value predictions. Softmax is a normalizing function applying a Boltzmann distribution to the inputs [4; 5; 13] such that all resulting values sum up to 1. Equation 15 defines the Softmax function for multi-label classification [12]. \hat{p}_k is the per-class probability of bearing the highest value. $s_k(x)$ denotes the score of class k for instance x , i.e., the output of the previous layer `q_value_prediction` at index k . The output then contains probabilities for each class to be of highest value, i.e., the probability for each input news item to receive a high reward upon recommending them to the user [20].

$$\hat{p}_k = \frac{\exp(s_k(x))}{\sum_{j \in K} \exp(s_j(x))} \quad (15)$$

Second, categorical cross-entropy performs loss calculation with the *cross-entropy cost function* presented in Equation 16 [12]. Here, Θ is the parameter set of the neural network we are training, thus, we search for an optimal Θ to minimize the cross entropy cost function $J(\Theta)$. Θ is a matrix containing parameter vectors $\theta^{(k)}$ for each class k that could be predicted, i.e., for each candidate that could be recommended. Equation 16 already accounts for training batches by averaging cross entropy loss over all m examples in the batch. For each example, the cross entropy is calculated based on true label $y^{(i)}$ and predicted probability $\hat{p}^{(i)}$ per class k .

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(\hat{p}_k^{(i)}) \quad (16)$$

In offline pre-training, where our training data only contains clicked items, i.e., only one item during each prediction should be predicted, we opt for the categorical cross-entropy loss function. The categorical loss entropy, or multi-class log loss, is optimized for cases where only one class should be predicted, i.e., multi-class classification problems. Categorical cross-entropy excels at multi-class classifications, because Softmax boosts the highest prediction and relaxes all

other predictions. For optimization of Θ , we perform training of the whole DDQN with categorical cross entropy loss function and the *Adam* optimizer. Adam is a robust and efficient method non-convex optimization problems in machine learning [22].

3.5.2 Binary Cross-Entropy Loss

In cases, where more than one item should be predicted, i.e., multi-label prediction, Softmax negatively influences performance. In multi-label classification, all predictions should be considered independently of one another. Thus, we use Tensorflow’s `binary_crossentropy` loss function.

Binary cross-entropy first applies a *Sigmoid*, or logistic, activation to the network’s outputs. Equation 17 shows, that calculation of each class’ probability \hat{p}_k to be predicted only depends on the score $s_k(x)$ for that class from the previous layer `q_value_prediction` and is independent from all other classes (c.f., $s_j(x)$ in the Softmax calculation in Equation 15) [12].

$$\hat{p}_k = \frac{1}{1 + \exp(-s_k(x))} \quad (17)$$

Next, binary cross-entropy calculates the loss using a *logistic regression cost function* given in Equation 18, sometimes called “log loss” [12]. Similar to the cross-entropy cost function, this function averages all examples i in a batch of size m to minimize the networks parameters θ . In contrast to the cross-entropy loss, we only have one vector of parameters θ in log loss instead of a matrix Θ of per-class parameters $\theta^{(k)}$.

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)})] \quad (18)$$

In online training, i.e., when the DDQN agent is deployed to ZDFMEDIATHEK, we expect multiple user clicks on a single list of recommendations. Thus, we opt for the binary cross-entropy loss function, which supports this kind of multi-label classification.

3.6 Reward

As defined in the MDP, our DDQN agent receives a reward for every performed recommendation. With this reward, the agent is retrained towards a potentially higher-valued policy. Thus, the reward depends on the agent’s success in recommending useful items, and the loss function assessing the DQN’s performance.

For effective DQN training, we model the reward as a tensor of the network’s output shape. Hence, we assign a reward to each predicted Q-value. As we define two different loss functions for offline and online training, we also specify distinctive offline and online rewards.

Furthermore, we aim at increasing user activeness, which we model as long term reward to be maximized by the DDQN agent. However, the agent cannot influence user activeness in offline training. Thus, user activeness only contributes to the online reward.

3.6.1 Offline Reward

In offline pre-training, where the click logs only provide one clicked item per record, we opted for the categorical cross-entropy loss function. Thus, all predictions of the DQN are normalized by a

Softmax function such that the sum of all predicted Q-values is 1. In the reward designated for offline training, we need to respect that Softmax normalization. Therefore, we create *true labels* from the click logs to train the network, which take the place of rewards in the offline case. Here, we simply provide a one-hot encoded true label that is 1 at the clicked news candidate index and 0 for all other news candidates. This ensures that the sum of all values in the true label always equals 1 to account for the Softmax normalization in categorical cross-entropy. Specifically, we online define this immediate reward from user clicks r_{click} in offline training. Thus, the total reward is trivially defined as follows:

$$r_{total} = r_{click} \quad (19)$$

3.6.2 Online Reward

In the online case, we aim at increasing long-term user satisfaction in addition to immediate rewards from clicks. As user satisfaction is not directly measurable, we calculate a *user activeness* score, that increases for highly active users and decreases for inactive users. Incorporating user activeness into the reward ensures that the agent develops a recommendation strategy benefiting the users in the long term.

User Activeness

We aim at increasing long-term benefits from recommendations by maximizing user activeness. Usually, users send multiple requests to a news platform in a short period of time, before leaving for several hours or days [60]. We consider each user request as *user return* to the platform and aim at increasing the amount of user returns. Therefore, we calculate user activeness by applying survival models to predict user return [17; 18; 37]. The *hazard function* given in Equation 20 denotes the rate of instantaneous user return at time t given that the user returns at time T and did not return until t , i.e., $T \geq t$ [60].

$$h(t) = \lim_{dt \rightarrow 0} \frac{Pr\{t \leq T < (t + dt) \mid T \geq t\}}{dt} \quad (20)$$

With this hazard function, we define the *survival function*³ $S(t)$, calculating the probability of user return after time t [37; 60]:

$$S(t) = \exp\left(-\int_0^t h(\tau) d\tau\right) \quad (21)$$

Next, we calculate the expected time until user return T_0 from Equation 21 [37; 60]:

$$T_0 = \int_0^\infty S(t) dt \quad (22)$$

We set a constant probability of user return $h(t) = h_0$ and update $S(t) = S(t) + S_a$ on each user return [60]. Furthermore, we adopt parameters suggested by Zheng et al. [60] based on real user return patterns:

³ Originally, the survival function denotes the probability of survival beyond time t given a rate of instantaneous death $h(\cdot)$.

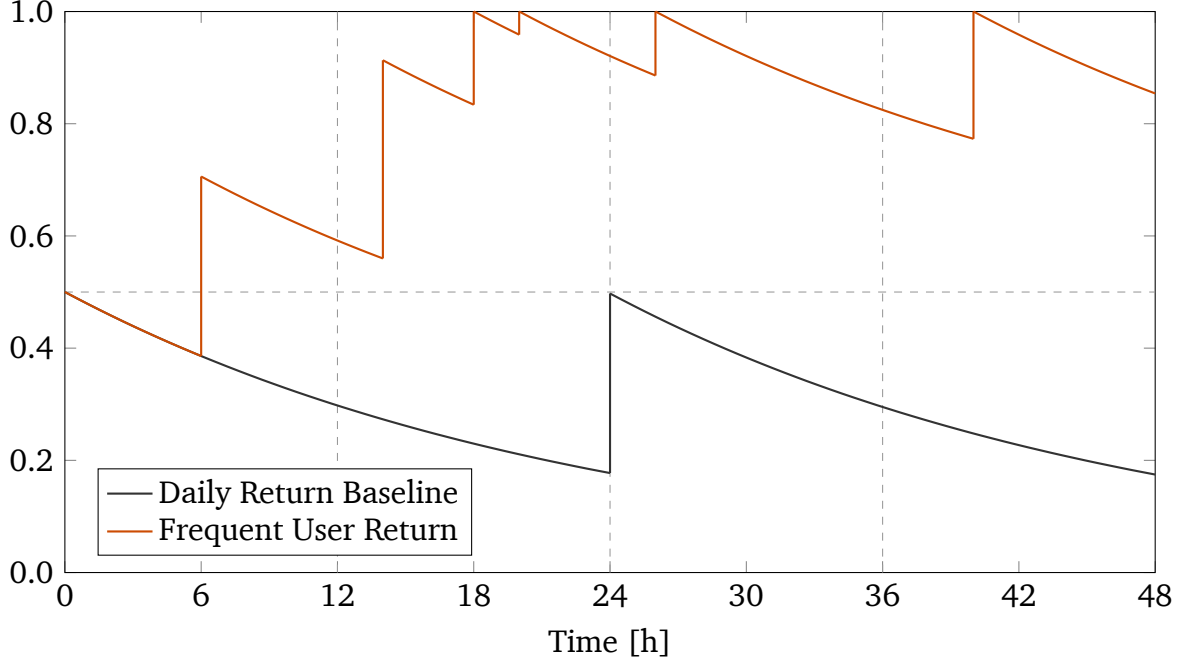


Figure 14: User Activeness Estimation for a User

$S_0 = 0.5$ denotes random initial probabilities of user return.

$T_0 = 24$ hours marks a daily user return to the news platform.

$h_0 = 1.2 \times 10^{-5} \text{s}^{-1}$ is calculated from Equation 21 and Equation 22.

$S_a = 0.32$ increases the calculated user activeness by 0.32 on each user return, ensuring that a daily click returns the user to the initial state, i.e., $S_0 = S_0 \exp(-h_0 T_0) + S_a$.

Next, we clip user activeness by $\{0, 1\}$ to prevent underflow into negative activeness and overflow in short sessions of high-frequency requests. Figure 14 shows the resulting user activeness estimation for an exemplary user compared to the baseline of one daily visit. Although the user visits more frequently after 18 hours, the user’s activeness is truncated to 1.

Online Reward

Since we aim at maximizing both immediate and future rewards, i.e., click through rates and user activeness, we combine click rewards and rewards from user activeness:

$$r_{total} = r_{click} + \beta r_{active} \quad (23)$$

Here, r_{click} may contain more than one user click for a request. Thus, r_{click} is multi-hot encoded, i.e., one-hot encoded for every clicked news candidate. β weighs future rewards, which Zheng et al. [60] set to 0.05, and $r_{active} = S(t)$ is the scalar activeness score added to each component of r_{click} . Since components of r_{click} are either 0 or 1, and the activeness score r_{active} and β are in $\{0, 1\}$, individual components of the resulting r_{total} are in $\{0, 2\}$.

In online training, we opted for the binary cross-entropy loss function normalizing each Q-value independently to $\{-1, 1\}$ through a Sigmoid function. Therefore, we clip all rewards in the online case per news candidate such that each Q-value lies in $\{-1, 1\}$. Instead of normalization, clipping does not decrease rewards for specific items, thus, conserving rewards for lower-rated items.

3.7 Exploration

Training the aforementioned model with gradient descent enforces exploitation of experience, i.e., the model always generates predictions bearing high rewards in the past. In the short term, this may boost model performance, but it renders predictions ineffective in dynamic environments. Therefore, we apply exploration to gain experience from new or less recommended items and prevent formation of a filter bubble for the user.

3.7.1 Epsilon-Greedy

ϵ -greedy is a simple exploration strategy that randomly selects items with a probability of ϵ . This approach effectively pursues exploration using only very limited resources. ϵ -greedy exploration usually starts with a high ϵ_0 of 1.0, i.e., the system recommends only random items at the beginning of training to quickly build up varying experience. During training, ϵ is gradually annealed to 0.05 over a certain number of steps. More specifically, at each step $i \in \{0, n - 1\}$ when annealing over n steps, the ϵ -greedy strategy updates $\epsilon_{i+1} = \epsilon_i - \frac{\epsilon_0}{n}$ until the pre-defined final ϵ_n is reached. Thus, for all steps $i \geq n$ the system only recommends random items in 5% of the requests, which commonly shows good performance [12].

We implemented an ϵ -greedy exploration strategy as described above, although it does not benefit the base model in the long term, i.e., ϵ -greedy does not improve our DDQN agent itself. Thus, our main approach features an exploration network in Dueling Bandit Gradient Descent as proposed by Zheng et al. [60] explicitly for news recommendation.

3.7.2 Dueling Bandit Gradient Descent

Zheng et al. [60] propose a new exploration strategy for the DDQN approach improving over the trivial ϵ -greedy approach. Instead of randomly selecting items for recommendation, an exploration network proposes items for recommendation.

Exploration Network

The exploration network is a clone of the Q-network described above, with altered weights. Equation 24 shows how the exploration network's weights are disturbed based on the Q-network's weights W . Weights W are multiplied by a random number between -1 and 1 and an *explore coefficient* $\rho \in \{0, 1\}$.

$$\Delta W = \rho \cdot \text{rand}(-1, 1) \cdot W \quad (24)$$

To receive the exploration network weights \widetilde{W} , we finally add the random ΔW to the original weights W in Equation 25 and assign these weights to the exploration network.

$$\widetilde{W} = W + \Delta W \quad (25)$$

Upon each request for recommendations, the DDQN agent then predicts a list of items L with the Q-network that is ranked by the predictions from the target network as described before. Additionally, the exploration predicts a second list of items \widetilde{L} . The agent then applies probabilistic interleave to merge both lists [16]. The resulting list contains the highest-ranked items from both L and \widetilde{L} which is presented to the user as recommendation list [60].

Probabilistic Interleave

In general, interleaving covers merging multiple lists into a single list. Contrary to alternative approaches such as balanced interleave or team draft, probabilistic interleave does not directly merge two lists L and \tilde{L} . Instead, probabilistic interleave constructs the interleaved list from two Softmax functions s and \tilde{s} produced by two rankers. Softmax ensures that all items from L and \tilde{L} have non-zero probability of being selected by either ranker and smoothes the expected rewards for all items in the lists. Here, Hofmann et al. [16] propose a Softmax function based on the reversed power of the rank of each item d in list L from the combined set of items $D = L \cup \tilde{L}$, as defined in Equation 26.

$$\hat{p}_d = \frac{r(d)^{-\tau}}{\sum_{d' \in D} r(d')^{-\tau}} \quad (26)$$

This variant of Softmax boosts valuation of top ranks and decreases latter ranks to emphasize the importance of recommending highly-rated items. τ controls how quickly probabilities decrease over the ranks, which Hofmann et al. [16] set to 3. Probabilistic interleave applies this Softmax function to all items in L and \tilde{L} . For each rank of the interleaved list, one of the resulting Softmax functions s or \tilde{s} is randomly chosen and provides an item d based on its ranking, while the assignment of either s or \tilde{s} to this rank is saved. Afterwards, the chosen item is removed from both lists L and \tilde{L} . Then, s and \tilde{s} are calculated again, based on the smaller lists, until the interleaved list reaches the desired length.

By saving the assignment of s or \tilde{s} to each rank of the resulting list, evaluation properly attributes later clicks on the corresponding items to the correct ranker, i.e., to the DDQN or exploration network in our case.

Dueling Bandit Gradient Descent

At training time, we compare whether items from exploitation list L or exploration list \tilde{L} received more user clicks. If items from the original list L received more clicks, we keep the Q-network as-is. In contrast, we update the Q-network towards the exploration network according to Equation 27 if items from the exploration network's list \tilde{L} received more user clicks. We calculate new weights W' for the Q-network based on the original weight W , the exploration weights \tilde{W} and a *exploitation coefficient* $\eta \in \{0, 1\}$ controlling the blending.

$$W' = W + \eta \tilde{W} \quad (27)$$

To sum up, the exploration network acts as a dueling agent to the DDQN. By merging the weights towards the improved network, the agent performs gradient descent over the networks' prediction performances or the received rewards respectively. Combined, we call this Dueling Bandit Gradient Descent (DBGD).



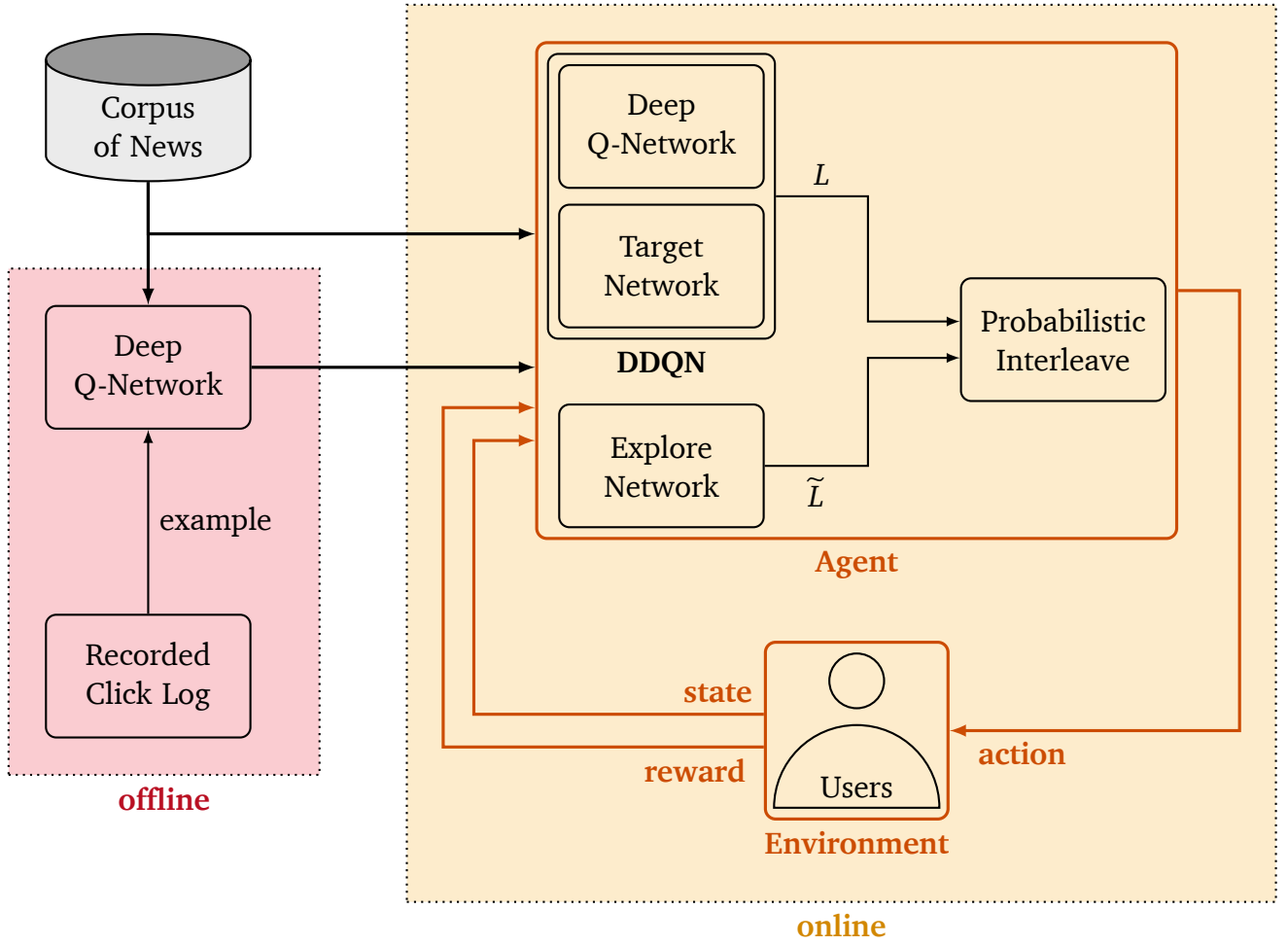


Figure 15: Conceptual View on the Technique in **MDP**

4 Implementation

We implemented the Double Dueling Deep Q-Network (DDQN) as described in section 3 using Python, machine learning frameworks in Python such as Numpy, Pandas and Scikit-learn, and the machine learning API Keras on top of the Tensorflow backend.

Specifically, we split implementation into offline simulation and online real-world phase as shown in Figure 15, where an offline simulator environment uses historical data to simulate an online scenario for training and evaluation purposes, and an online real-world environment covers live user interactions in ZDFMEDIATHEK. Thus, the simulator environment allows for an agent in offline training to act as if it were in online training, which replaces the *offline* part of Figure 15 with a simulated *online* part.

Furthermore, we implemented multiple agents covering different or all parts of our DDQN approach and ϵ -greedy or Dueling Bandit Gradient Descent (DBGD) exploration strategy: an offline agent that directly trains the Q-network on training examples, and an online agent with DDQN that either applies *varepsilon*-greedy exploration or DBGD with an exploration network.

```
# Use tensorflow:1.14 as base image
FROM tensorflow/tensorflow:1.14.0-gpu-py3

# install most python dependencies via pip
RUN pip3 install --user numpy scipy matplotlib ipython jupyter pandas \
    sympy nose scikit-learn dataclasses tqdm pydot ml_metrics flask pulp

# install git and graphviz via apt
RUN apt-get update -y
RUN apt-get install git graphviz -y

# install python dependencies via git
RUN pip3 install --user git+https://github.com/mpkato/interleaving.git
```

Figure 16: Dockerfile preparing our Tensorflow docker image

4.1 Frameworks

We implemented the DDQN described in section 3 in Keras⁴ for Python, a high-level API for machine learning tasks leveraging the Tensorflow backend⁵. More specifically, we targeted Python 3.7, Tensorflow r1.14 and the accompanying *tf.Keras* module, a Keras implementation directly included in Tensorflow. Instead of a manual Tensorflow installation, we used the recommended tensorflow-gpu docker image, which we extended to our needs through the dockerfile provided in Figure 16. Naturally, this dockerfile extends the *tensorflow:1.14.0-gpu-py3* docker image, which is the GPU-capable, Python3-compatible version of the Tensorflow 1.14 docker image, and installs several python libraries via python’s PIP and the graphviz visualization tool via Ubuntu’s APT package manager. We use these additional python libraries for extended preprocessing outside of the Tensorflow context and graphviz for creation of the architectural overview shown in Figure 12. This dockerfile enables the docker build command to correctly build our image, e.g., via

```
| $> docker build -t vkuhn/tenserflow-extended:1.14.0-gpu-py3 .
```

tagging the new image as *vkuhn/tenserflow-extended:1.14.0-gpu-py3*.

The defined Q-network is summarized by Keras in Figure 17. In total, this network has 47,638,217 (trainable) parameters, i.e., layer weights and biases. Furthermore, Figure 17 describes the layer’s dimensions and connections, i.e., previous layer to each layer. This summary is ultimately identical to the network architecture presented in section 3. The target network and exploration network share the exactly same definition with distinct weights.

⁴ Keras: <https://keras.io/>

⁵ Tensorflow: <https://www.tensorflow.org/>

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
user (InputLayer)	[(None, 1920)]	0	
context (InputLayer)	[(None, 200, 38)]	0	
interactions (InputLayer)	[(None, 20, 20)]	0	
news (InputLayer)	[(None, 200, 367)]	0	
flat_user (Flatten)	(None, 1920)	0	user[0][0]
flat_context (Flatten)	(None, 7600)	0	context[0][0]
flat_interactions (Flatten)	(None, 400)	0	interactions[0][0]
flat_news (Flatten)	(None, 73400)	0	news[0][0]
advantage_features (Concatenate)	(None, 83320)	0	flat_user[0][0] flat_context[0][0] flat_interactions[0][0] flat_news[0][0]
hidden_advantage_prediction (Dense)	(None, 512)	42660352	advantage_features[0][0]
dueling_state_features (Concatenate)	(None, 9520)	0	flat_user[0][0] flat_context[0][0]
lrelu_advantage_prediction (LeakyReLU)	(None, 512)	0	hidden_advantage_prediction[0][0]
dueling_hidden_state (Dense)	(None, 512)	4874752	dueling_state_features[0][0]
advantage_prediction (Dense)	(None, 200)	102600	lrelu_advantage_prediction[0][0]
dueling_value_prediction (Dense)	(None, 1)	513	dueling_hidden_state[0][0]
q_value_prediction (Lambda)	(None, 200)	0	advantage_prediction[0][0] dueling_value_prediction[0][0]
Total params: 47,638,217			
Trainable params: 47,638,217			
Non-trainable params: 0			

Figure 17: Keras Summary of the Q-network

Input: Click logs L and news corpus C provided by ZDF_{MEDIATHEK},
length of user history considered for recommendation h ,
amount of candidates used for recommendation c , and
length of recommendation list k

Output: Simulator memory M

Initialize $M = \{\}$;

for $l \in L$ **do**

 Observe current state S ;

 Observe request time $l.t$;

 Observe request user $l.u$;

 Collect user interactions $I = \{i_1, \dots, i_h\} \subset L$;

 Build user feature $U = \{\sum_{1_hour} i, \sum_{6_hours} i, \sum_{day} i, \sum_{week} i, \sum_{year} i : i \in L\}$;

 Select news candidates $N = \{n_1, \dots, n_c\} \subset C$;

 Calculate context for the candidates $X = \{x_1, \dots, x_{|N|}\}$;

 Predict q-values for the candidates $Q = \{q_1, \dots, q_{|N|}\}$;

 Calculate reward from prediction and click log $R = \{r_1, \dots, r_{|N|}\}$;

 Add $((S, (I, U, N, X) \rightarrow Q) \rightarrow R)$ to M ;

return M

Figure 18: Building simulator memory

4.2 Environment

We implemented two environments: a simulator environment for replaying historical click logs and a real-world environment for live experiments in ZDF_{MEDIATHEK}.

Simulator Environment

First, we implemented a simulator environment for Reinforcement Learning (RL) to train and test our DDQN. Contrary to Shi et al. [46], we did not extend the Gym engine⁶, which is a toolkit for developing and comparing RL algorithms which provides much overhead and requires an additional installation currently not compatible with Microsoft Windows [6]. Instead, we focused on simplicity and portability by building a simulator for RL that iterates historical click logs provided by ZDF_{MEDIATHEK}, requests for recommendations and checks whether the actually clicked news is included in the recommendations list as shown in Figure 18 in an approach similar to Zhao et al. [58].

Figure 18 describes how the agent’s memory is built in the simulator environment. The simulator has access to all historical click logs, the corpus of news and the lengths of provided user history and candidates used for recommendation. For each entry in the click logs, the simulator observes the state, time of request and user id. Based on these observations, the simulator builds all features required for the DDQN as described in section 3: a set of historical user interactions, a user feature, a set of news candidates, and, a set of context features. The simulator feeds these features into the DDQN to receive predicted q-values for all candidates. Afterwards, the simulator calculates a reward based on how the actually clicked item is ranked in the prediction.

⁶ OpenAI Gym: <https://github.com/openai/gym>

All the features, prediction, reward, and, current state are then saved in the memory for later memory replay.

Real-World Environment

Second, we implemented a real-world RL environment, which receives recommendation requests via HTTP and responds with the predicted recommendation. Each request is temporarily saved into memory. Afterwards, clicks are collected and appended to the requests in memory to be used as rewards. With this information, the DDQN is periodically retrained, i.e., each hour, to stay up-to-date with changing user interests and the dynamic corpus of news. Unfortunately, we did not connect this environment with ZDFMEDIATHEK in time for this thesis. Thus, we are evaluating on the simulator environment only.

4.3 Agent

Additionally, we designed two agents: an offline⁷ agent which fits the network to the click logs instead of performing RL, and, an online agent which acts according to our technique presented in section 3. While the online agent is required in an actual recommendation scenario within the real-world environment, pre-training of the DDQN weights may be performed either using the online or offline agent within the simulator environment.

Offline Agent

The offline agent solely trains the Q-network via Keras' `fit_generator` function for training a model on training data. Specifically, this offline agent trains the network for pure exploitation and does not apply any exploration strategy. Although pure exploitation may harm model performance, the offline agent is by far the simplest approach implementation-wise and profits most from the optimized high-level API to model specification and training of Keras.

Online Agent

The online agent performs RL for the DDQN approach described in section 3. Besides, the online agent either applies ϵ -greedy exploration or creates an exploration network for DBGD. For RL and exploration, the online agent requires much more “manual” Python training code than the offline agent, making it large and potentially inefficient.

⁷ Machine learning terminology: data becomes available sequentially in “online” learning compared to the entire data being available in “offline” learning



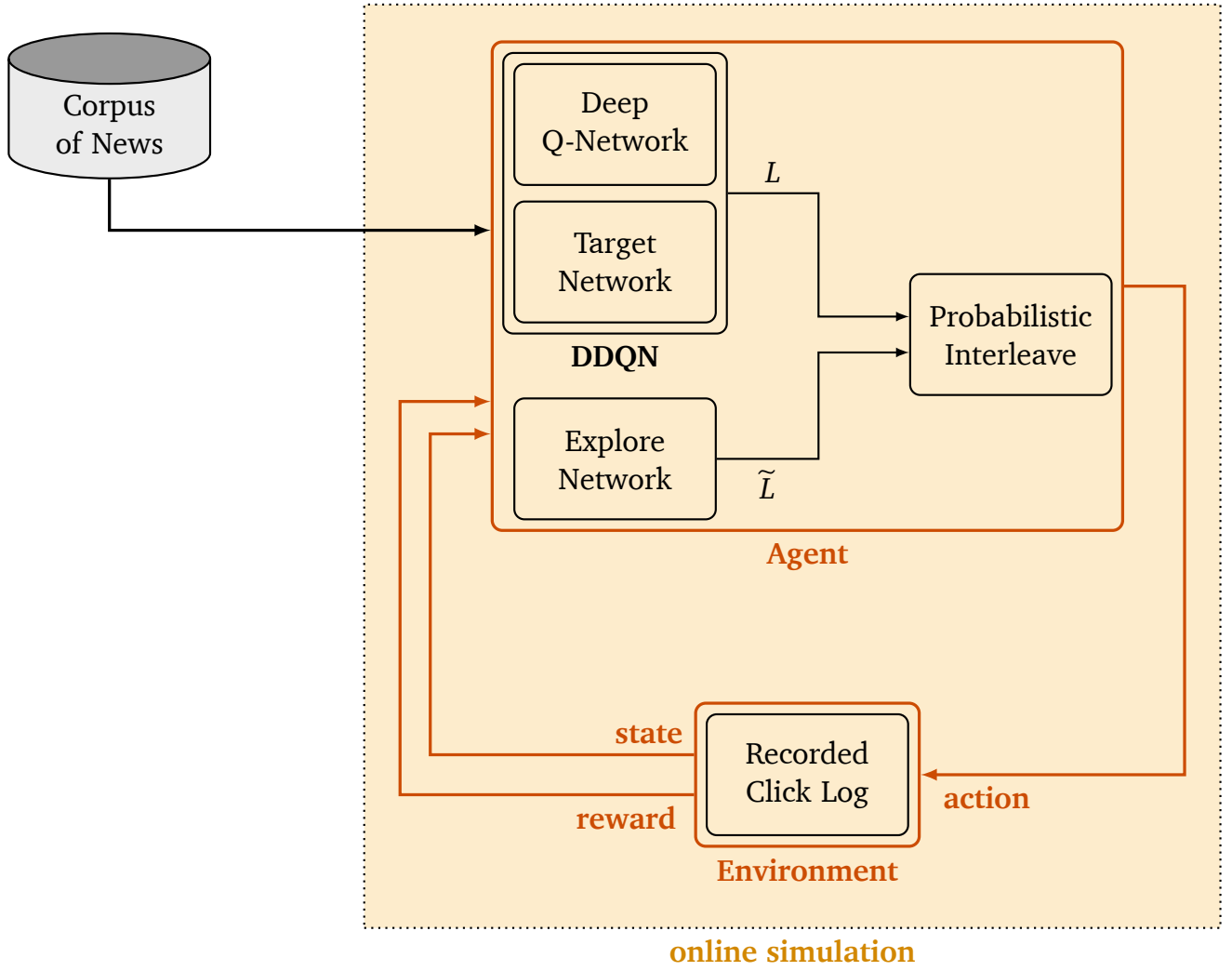


Figure 19: Conceptual View on the Simulator Environment

5 Evaluation

In this evaluation, we concentrate on the offline environment as we did not conduct experiments with the real-world environment in time for this thesis. Thus, we test three different models on offline data in the simulator environment and compare their performance in simulated Reinforcement Learning (RL): the offline agent, an online agent with Dueling Bandit Gradient Descent (DBGD), and, an online agent with ε -greedy exploration. For the latter two online agents, the simulator behaves as if the agents acted online, i.e., presenting rewards from the simulator environment as shown in Figure 19.

Since the agent’s recommendations do not influence user behavior in click log simulation, we resort to the offline reward defined in section 3. Specifically, this reward omits future user activeness, i.e., we only evaluate immediate rewards based on user clicks simulated by the simulator environment.

First, we provide information about our setup of software and hardware used for evaluation. Next, we defined the hyperparameters set for evaluation and describe our data and applied split for training, validation and testing. Finally, we present evaluation results of the three evaluated models.

General Hyperparameters			
Parameter	Description	Value	Note
α	Learning rate of Adam	0.001	Original value [22]
γ	Discount factor of future rewards	0.4	Original value [60]
c	Number of news candidates per request	200	Balance diversity & efficiency
h	Number of interactions history per request	20	ZDFMEDIATHEK provides 20 interactions
$ L $	Number of items recommended per request	10	Common value for recommendation lists

Hyperparameters for ε -greedy Exploration			
Parameter	Description	Value	Note
ε_0	Initial probability of random exploration	1.0	Provided in [12, p. 632]
ε_n	Final probability of random exploration	0.05	Provided in [12, p. 632]
n	Steps to anneal initial to final ε	39836	20% of our training data

Hyperparameters for Dueling Bandit Gradient Descent			
Parameter	Description	Value	Note
ρ	Exploration coefficient	0.1	Original value [60]
η	Exploitation coefficient	0.05	Original value [60]
$ \tilde{L} $	Number of exploration items	10	$ \tilde{L} = L $ for probabilistic interleave

Table 1: Evaluation Hyperparameters

5.1 Experiment Setup

All experiments were conducted on a server with a Ubuntu 18.04 LTS installation featuring 256 GB of RAM and a Titan X (Pascal) graphics card with 12GB of on-board memory. As described in section 4, we executed the DDQN in our docker image:

```
$> docker run --gpus all -d -it --rm -v ./zdf_dqn:/tmp \
-w /tmp/zdf_dqn_keras vkuhn/tensorflow-extended:1.14.0-gpu-py3 \
python3 -m zdf_dqn.main --train_dqn --test_dqn --multiprocessing
```

For the different approaches in our evaluation, we only appended few parameters: `--offline` to stay offline in a networking sense, i.e., to use our simulator environment, `--fit` to use the Keras `model.fit_generator` function, i.e., to choose the offline agent, and `--e-greedy` to use ε -greedy exploration instead of DBGD.

5.2 Hyperparameters

The models we evaluated have different hyperparameters, i.e., parameters affecting the training or exploration. In contrast to the parameters of the model such as layer weights, hyperparameters are set beforehand and partially determine training and prediction performance. For almost all hyperparameters, we set defaults recommended by papers introducing or evaluating the respective approaches. We split the hyperparameters into three sets: general hyperparameters, hyperparameters specific to ε -greedy exploration, and, hyperparameters specific to the exploration network in DBGD. Table 1 gives an overview over all hyperparameters and presents their values in the evaluation.

General hyperparameters affect all evaluated models. Here, we set Adam’s learning rate $\alpha = 0.001$ to its original value introduced by Kingma and Ba [22] and the discount factor for future

rewards γ to 0.4 as provided by Zheng et al. [60]. Additionally, we define the final model size here by setting values for the amount of *candidates*, *interactions* and *recommended items* L , such that they fit the ZDFMEDIATHEK domain and API requirements and offer enough diversity in the candidates while keeping the model sufficiently small and efficient to be computationally feasible on the provided hardware. Next, we define hyperparameters for ε -greedy exploration. Annealing ε from 1.0 to 0.05 has proven good performance in practice [12]. We reach the final ε after 39836 examples, which equals 20% of our training set. Furthermore, we define the exploration coefficient ρ and exploitation coefficient η for DBGD of the Double Dueling Deep Q-Network (DDQN) and exploration network as introduced by Zheng et al. [60]. Besides, the exploration network should generate recommendations of the same length as the DDQN, i.e. $|\tilde{L}| = |L| = 10$.

5.3 Data

For training and evaluation, we use sampled data provided by ZDFMEDIATHEK. The sample contains 72147 news items and 311221 clicklog entries. The clicklog entries were collected between June 19, 2019 and June 22, 2019, i.e., the sample contains data of four consecutive days. We split this data into three subsets: the chronologically last 20% are reserved for testing. Afterwards, we split the first 80% again into 80% train set and 20% validation set. The created train set contains the chronologically first 199180 logged clicks, the validation set consists of the next 49795 clicklog entries and the test set resembles the remaining 62246 clicks.

Although cross-validation helps better estimate generalization of our model to unknown data and, thus, productive model performance, we chose to chronologically split the data [23]. We decided on a chronological split for better simulation of the RL task. This enables the model to directly demonstrate the performance of its RL capabilities during training, validation and testing phase.

Before feeding data into the network(s), we remove all text columns from the data, and apply one-hot encoding on categorical columns as described in section 3.

5.4 Experiment Results

We determine the performance of all evaluated approaches by means of Mean Reciprocal Rank (MRR), Discounted Cumulative Gain (DCG) and Top-k Categorical Accuracy.

The reciprocal rank evaluates, at which position in a list the first relevant item is placed. Here, relevant items are those items clicked by the user. For the MRR, the reciprocal rank is evaluated for a set of *queries* Q , which are requests for recommendations in our scenario. The MRR is defined as the mean of the sum of all individual reciprocal ranks for queries in Q , i.e.:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^Q \frac{1}{rank_i} \quad (28)$$

Here, $rank_i$ denotes the rank of the first relevant item in the recommendation list for query i .

The cumulative gain evaluates relevance of all results in an item list, which is discounted by the rank of the items in DCG. We define DCG as

$$DCG_p = \sum_{i=1}^p \frac{rel_i}{\log_2(i+1)} \quad (29)$$

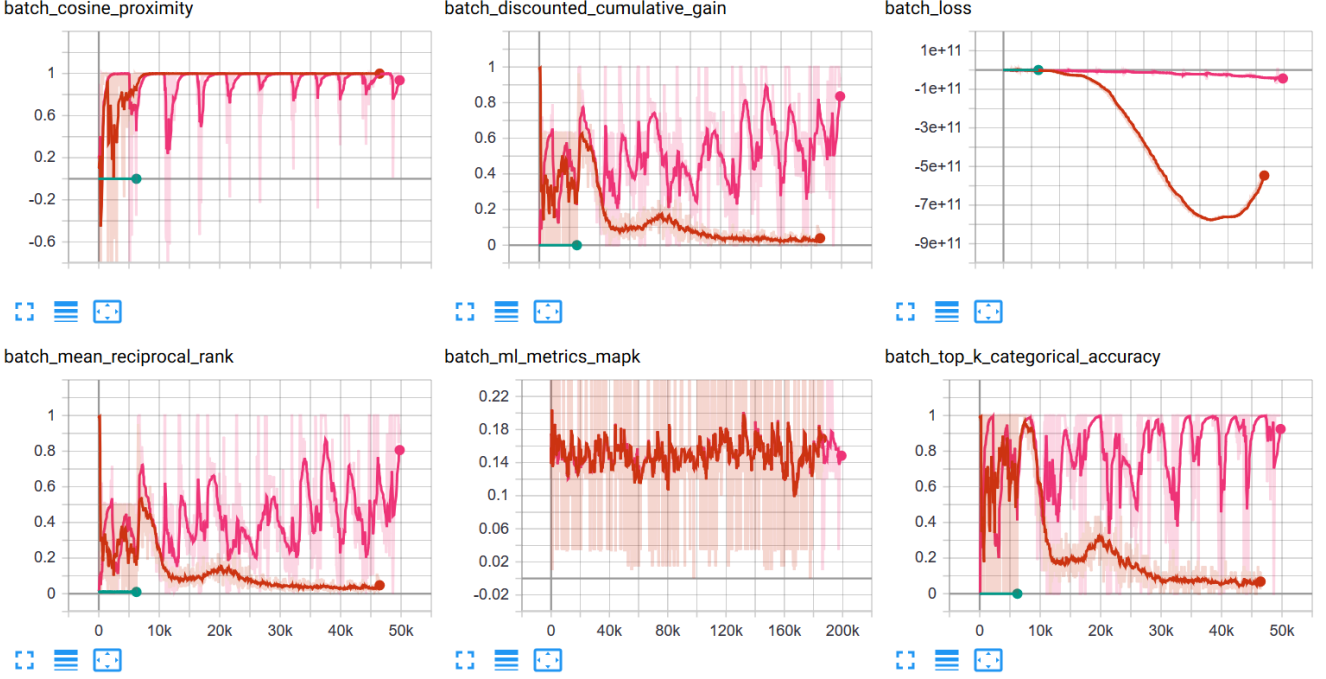


Figure 20: Training Metrics: CP, DCG, Loss (CCE), MRR, MAP@10, Top-10 Cat. Acc.

where i marks the position of an item in the recommendation list, rel_i is the graded relevance of the item at position i and p denotes the number of items that are considered in DCG calculation. We set p to 10 as we are showing 10 out of the 200 news candidates to the user. For graded relevance, we set 1 on items they user clicked and 0 otherwise.

The Top-k Categorical Accuracy is a categorization metric that measures whether the target class is present among the first k items of a prediction. We define the clicked item of a clicklog entry as target class for this metric and, again, set k to 10 to account for the length of recommendation lists in our setup. Since we only have one clicked item in the click log at each time, we may use this metric in the simulator scenario.

Figure 20 gives an overview of various metrics and the loss during training. The different counts on the x-axis are due to implementation differences, that we did not resolve yet. However, these graphs still provide an overview of the training performance of our evaluation models. Green marks the offline agent, orange the ϵ -greedy online agent and pink denotes the DBGD online agent. Strangely, the offline agent’s loss and metrics are always exactly 0, so we suspect that our implementation of the Keras logging interface *Tensorboard callback* is incomplete.

However, for the ϵ -greedy and DBGD approaches, we notice trends: the agent with ϵ -greedy exploration (orange) minimizes the categorical cross-entropy loss in the upper right of Figure 20 quicker and shows a more stable cosine proximity of prediction and true labels. Nevertheless, this ϵ -greedy agent also decreases DCG, MRR, and, Top-k Categorical Accuracy during training, which should increase towards 1. In contrast, the DBGD agent (pink) improves the latter three metrics and shows similar cosine proximity, although its loss decreases only slightly. The jumps in the cosine proximity of the DBGD agent most likely mark the times of exploration network reset after Q-network retraining. Besides, we added a plot for Mean Average Precision (MAP) at k (MAP@ k), which is calculated by a Python library and, thus, cannot be directly included as Tensorflow metric. Therefore, we did not calculate MAP@ k for the offline agent, which is trained

Experiment Results							
Model	Training Time	Cosine Proximity	DCG	Categorical Cross-Ent.	MRR	MAP	Top-k Cat. Accuracy
Offline Fit	7h	0.00	0.00	0.00	0.00	-	0.00
ϵ -greedy	68h	0.94	0.14	-3.47e11	0.12	0.15	0.26
DBGD	156h	0.92	0.51	-1.53e10	0.42	0.15	0.80

Table 2: Experiment Results

by Keras `fit_generator` method that only evaluates Tensorflow metrics efficiently. MAP@k with $k = 10$ is the mean average precision of the top 10 values in a prediction, which exhibits an equally stagnating trend for both ϵ -greedy and DBGD respectively.

These trends are also visible in the means of each metric over the training time. Table 2 provides the training times, and means of all metrics presented in Figure 20. Here, bold results represent the best result within each column. Again, we notice very similar trends in Cosine Proximity and MAP@10. In DCG, MRR and Top-k Categorical Accuracy, the approach utilizing DBGD is far superior, although the ϵ -greedy approach further minimizes the categorical cross-entropy loss.

Besides, Table 2 shows that training time of the ϵ -greedy is almost 10 times higher than training time of the offline agent. Furthermore, training time of the DBGD agent more than doubles that of the ϵ -greedy approach. Both online agents require more training time, as the training method of the offline agent is heavily optimized by Tensorflow. However, the exploration network and interleaving in DBGD adds additional computational complexity over the simple ϵ -greedy exploration strategy.

Testing on an independent test set is required to consolidate our results from training. Although these results seem to show improvement of DBGD over ϵ -greedy exploration, the improvements could stem from overfitting on our relatively small and temporally narrow training data. Unfortunately, a bug in our test script in combination with long training and testing times prevented the presentation of test metrics in this place, which remains as future work.



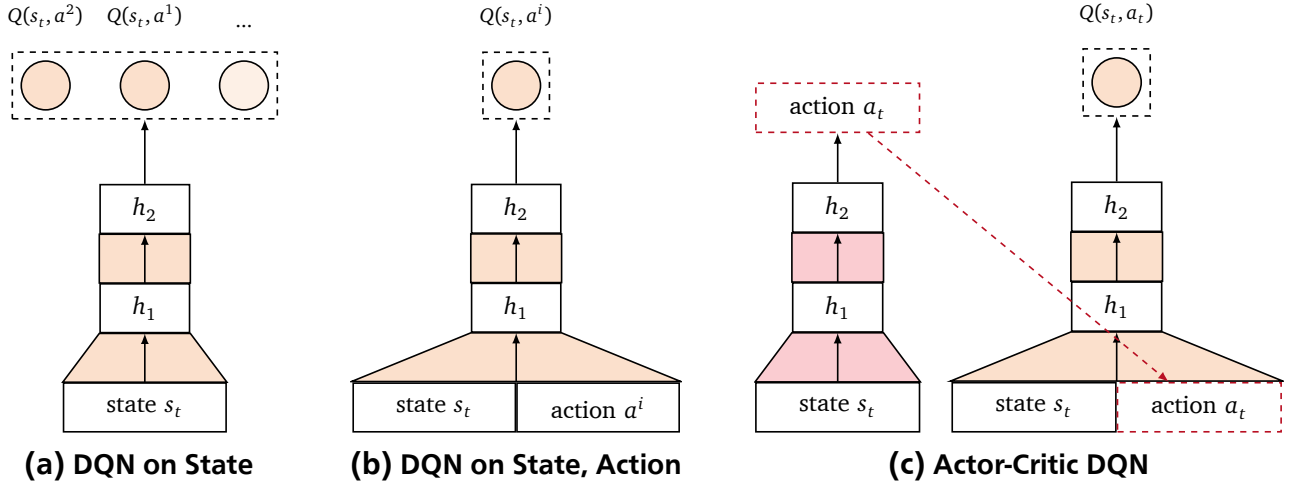


Figure 21: Comparison of Deep Q-Network Architectures [c.f. 58]

6 Related Work

Our technique is heavily based on the *DRN* approach by Zheng et al. [60]. Mostly, we recreated the Dueling Bandit Gradient Descent (DBGD) of Double Dueling Deep Q-Network (DDQN) architecture and exploration network from Zheng et al. [60] and adapted the networks to the features available in ZDFMEDIATHEK. In contrast to our evaluation, Zheng et al. [60] were able to conduct online experiments in a real-world news recommender platform, indicating that the DDQN approach with consideration of user activeness yielded better results in online recommendation accuracy and diversity of recommended items than various compared models: Logistic Regression (LR), Factorization Machines (FM), (Hidden) Linear Upper Confidence Bound, and, several compositions of DDQN, ϵ -greedy, and, DBGD.

Actor-Critic Architecture

Besides the naturally related work of Zheng et al. [60], we consider actor-critic approaches as closely related [31; 58]. While Zhao et al. [58] focus on comparison of Deep Q-Network (DQN) architectures to present a framework for list-wise recommendations (*LIRD*), Liu et al. [31] examine explicit user-item interactions modeling in their *DRR* method.

For instance, Zhao et al. [58] present three base architectures for Q-learning as shown in Figure 21. The first architecture, shown in Figure 21a, receives only state features and outputs Q-values for all possible actions. Therefore, this architecture is limited to small actions spaces. Our approach uses the second type of architecture as shown in Figure 21b, although we calculate 200 Q-values at once instead of only the Q-value for a single action. Here, the DQN operates on state and action features, but still has to calculate Q-values for all items in the action space. In contrast, the actor-critic architecture presented in Figure 21c features two networks: the *actor* network only decides on an action depending on the state and the *critic* network calculates a Q-value only for the chosen action.

Liu et al. [31] utilized the actor-critic architecture to explicitly model user-item interactions in a recommender system. In order to explicitly model interactions, Liu et al. [31] created a state representation from the user's interaction history, similar to the interaction inputs in our approach. Then, the actor network generates a ranking function a_t as action, which ranks

available items i_1, \dots, i_n by the scalar product $a^\top i_t$, where the item leading to the highest scalar product is recommended to the user. Afterwards, the critic network predicts a Q-value $Q(s_t, a_t)$ for the action a_t to update the actor network towards maximizing $Q(s_t, a_t)$.

In conclusion, the critic network is very similar to the target network in a DDQN approach. However, replacing the Q-network of a DDQN by an actor may improve performance and scalability on large data sets. Besides, Liu et al. [31] do not explicitly improve exploration. Thus, we believe integration of both approaches, i.e., DBGD of an actor-critic recommender and an exploration network, may benefit from the actors scalability and from the diversification of the exploration network alike.



7 Conclusion

We present a novel recommendation approach and exploration strategy based on the DRN framework [60]. The incorporated Dueling Bandit Gradient Descent (DBGD) with an exploration network promises better exploration and diversification in recommendations to prevent building of a filter bubble for users. Furthermore, explicitly considering future rewards such as user activeness should benefit long-term performance of the Reinforcement Learning (RL) agent. Unfortunately, we were not able to evaluate online performance in time for this thesis.

However, our offline experiments hint at improved performance of DBGD of the Double Dueling Deep Q-Network (DDQN) agent and an exploration network in terms of Discounted Cumulative Gain (DCG), Mean Reciprocal Rank (MRR), and, Top-k Categorical Accuracy compared to ϵ -greedy exploration. However, adding the exploration network vastly increases training time due to its high computational complexity and additional storage of interleaving results. Since we did not conduct reliable offline tests and online results to validate these results and negate the suspicion of overfitting, these results are to be handled with care until consolidated.

In the future, we aim to conduct online experiments in ZDFMEDIATHEK to collect more robust results about model performance. Besides, we plan to improve the training procedure for increased performance, including introduction of proper multiprocessing in the RL simulator part, upgrading towards the Tensorflow 2 API and transitioning to the Tensorflow Estimators API for stability and deployability. Furthermore, we intend to improve model performance by learning embeddings of categorical variables instead of one-hot encoding, or learn time-sensitive embeddings of the complete features [8]. This should result in a massively reduced model size and, thus, enable faster training and prediction on the one hand or allow more candidates to be fed into the model. Additionally, we plan to evaluate variable length inputs via LSTM layers, e.g., when a certain user has fewer interactions on record than expected by the interactions input layer. Furthermore, word embeddings on so far discarded columns such as news title and lead paragraph may result in improved recommendations by introducing more content similarity into the approach. For instance, fastText⁸ offers pre-trained word embeddings for a large vocabulary in 157 languages including German [15].

⁸ fastText: <https://fastText.cc>



Acronyms

ANN	Artificial Neural Network	9
CB	Content-Based	1
CF	Collaborative Filtering	1
DBGD	Dueling Bandit Gradient Descent	2
DCG	Discounted Cumulative Gain	37
DDQN	Double Dueling Deep Q-Network	2
DL	Deep Learning	2
DQN	Deep Q-Network	3
FM	Factorization Machines	5
LR	Logistic Regression	5
MAB	Multi-Armed Bandit	5
MAP	Mean Average Precision	38
MDP	Markov Decision Process	6
MF	Matrix Factorization	5
MLP	Multilayer Perceptron	9
MRR	Mean Reciprocal Rank	37
OTT	Over-The-Top	1
RL	Reinforcement Learning	2



References

- [1] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. „A brief survey of deep reinforcement learning“. In: *arXiv preprint arXiv:1708.05866* (2017).
- [2] Richard Bellman. „A Markovian decision process“. In: *Journal of mathematics and mechanics* (1957), pp. 679–684.
- [3] Hendrik Blockeel and Joaquin Vanschoren. „Experiment databases: Towards an improved experimental methodology in machine learning“. In: *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer. 2007, pp. 6–17.
- [4] Ludwig Boltzmann. „Studien über das Gleichgewicht der lebendigen Kraft zwischen bewegten materiellen Punkten“. In: *Wiener Berichte* 58 (1868), pp. 517–560.
- [5] John S Bridle. „Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition“. In: *Neurocomputing*. Springer, 1990, pp. 227–236.
- [6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.
- [7] Olivier Chapelle and Lihong Li. „An empirical evaluation of thompson sampling“. In: *Advances in neural information processing systems*. 2011, pp. 2249–2257.
- [8] Felipe Soares da Costa and Peter Dolog. „Collective embedding for neural context-aware recommender systems“. In: *Proceedings of the 13th ACM Conference on Recommender Systems*. ACM. 2019, pp. 201–209.
- [9] Paul Covington, Jay Adams, and Emre Sargin. „Deep neural networks for youtube recommendations“. In: *Proceedings of the 10th ACM conference on recommender systems*. ACM. 2016, pp. 191–198.
- [10] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. „Google news personalization: scalable online collaborative filtering“. In: *Proceedings of the 16th international conference on World Wide Web*. ACM. 2007, pp. 271–280.
- [11] Mukund Deshpande and George Karypis. „Item-based top-n recommendation algorithms“. In: *ACM Transactions on Information Systems (TOIS)* 22.1 (2004), pp. 143–177.
- [12] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.
- [13] Josiah Willard Gibbs. *Elementary principles in statistical mechanics developed with especial reference to the rational foundation of thermodynamics*. 1902.
- [14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. [http : / / www . deeplearningbook . org](http://www.deeplearningbook.org). MIT Press, 2016.
- [15] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. „Learning Word Vectors for 157 Languages“. In: *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*. 2018.
- [16] Katja Hofmann, Shimon Whiteson, and Maarten De Rijke. „A probabilistic method for inferring preferences from clicks“. In: *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM. 2011, pp. 249–258.

-
- [17] Joseph G. Ibrahim, Ming-Hui Chen, and Debajyoti Sinha. „Bayesian Survival Analysis“. In: *Encyclopedia of Biostatistics*. American Cancer Society, 2005. ISBN: 9780470011812. DOI: 10.1002/0470011815.b2a11006. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/0470011815.b2a11006>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/0470011815.b2a11006>.
- [18] How Jing and Alexander J Smola. „Neural survival recommender“. In: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM. 2017, pp. 515–524.
- [19] Yuchin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. „Field-aware factorization machines for CTR prediction“. In: *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM. 2016, pp. 43–50.
- [20] Frank Kane. „Building Recommender Systems with Machine Learning and AI: Help people discover new products and content with deep learning, neural networks, and machine learning recommendations.“ In: (2018).
- [21] Bitna Kim and Young Ho Park. „Beginner’s guide to neural networks for the MNIST dataset using MATLAB“. In: *The Korean Journal of Mathematics* 26.2 (2018), pp. 337–348.
- [22] Diederik P Kingma and Jimmy Ba. „Adam: A method for stochastic optimization“. In: *arXiv preprint arXiv:1412.6980* (2014).
- [23] Ron Kohavi et al. „A study of cross-validation and bootstrap for accuracy estimation and model selection“. In: *Ijcai*. Vol. 14. 2. Montreal, Canada. 1995, pp. 1137–1145.
- [24] Yehuda Koren, Robert Bell, and Chris Volinsky. „Matrix factorization techniques for recommender systems“. In: *Computer* 8 (2009), pp. 30–37.
- [25] Paul Lamere and S Green. „Project Aura: recommendation for the rest of us“. In: *Presentation at Sun JavaOne Conference*. 2008.
- [26] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.
- [27] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. „A contextual-bandit approach to personalized news article recommendation“. In: *Proceedings of the 19th international conference on World wide web*. ACM. 2010, pp. 661–670.
- [28] Yuxi Li. „Deep reinforcement learning“. In: *arXiv preprint arXiv:1810.06339* (2018).
- [29] Yuxi Li. „Deep reinforcement learning: An overview“. In: *arXiv preprint arXiv:1701.07274* (2017).
- [30] Greg Linden, Brent Smith, and Jeremy York. „Amazon.com recommendations: Item-to-item collaborative filtering“. In: *IEEE Internet computing* 1 (2003), pp. 76–80.
- [31] Feng Liu, Ruiming Tang, Xutao Li, Weinan Zhang, Yunming Ye, Haokun Chen, Huifeng Guo, and Yuzhou Zhang. „Deep reinforcement learning based recommendation with explicit user-item interactions modeling“. In: *arXiv preprint arXiv:1810.12027* (2018).
- [32] Zhongqi Lu and Qiang Yang. „Partially observable Markov decision process for recommender systems“. In: *arXiv preprint arXiv:1608.07793* (2016).
- [33] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. „Rectifier nonlinearities improve neural network acoustic models“. In: *Proc. icml*. Vol. 30. 1. 2013, p. 3.

-
- [34] Tariq Mahmood and Francesco Ricci. „Learning and adaptivity in interactive recommender systems“. In: *Proceedings of the ninth international conference on Electronic commerce*. ACM. 2007, pp. 75–84.
- [35] James L McClelland, David E Rumelhart, PDP Research Group, et al. *Parallel distributed processing*. Vol. 2. MIT press Cambridge, MA: 1987.
- [36] H Brendan McMahan et al. „Ad click prediction: a view from the trenches“. In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2013, pp. 1222–1230.
- [37] Rupert G Miller Jr. *Survival analysis*. Vol. 66. John Wiley & Sons, 2011.
- [38] Volodymyr Mnih et al. „Human-level control through deep reinforcement learning“. In: *Nature* 518.7540 (2015), p. 529.
- [39] Raymond J Mooney and Lorie Roy. „Content-based book recommending using learning for text categorization“. In: *Proceedings of the fifth ACM conference on Digital libraries*. ACM. 2000, pp. 195–204.
- [40] Steffen Rendle. „Factorization machines“. In: *2010 IEEE International Conference on Data Mining*. IEEE. 2010, pp. 995–1000.
- [41] Pornthep Rojanavas, Phaitoon Srinil, and Ouen Pinngern. „New recommendation system using reinforcement learning“. In: *Special Issue of the Intl. J. Computer, the Internet and Management* 13.SP 3 (2005).
- [42] Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.
- [43] Frank Rosenblatt. *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Tech. rep. Cornell Aeronautical Lab Inc Buffalo NY, 1961.
- [44] David E Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. „Backpropagation: The basic theory“. In: *Backpropagation: Theory, architectures and applications* (1995), pp. 1–34.
- [45] Guy Shani, David Heckerman, and Ronen I Brafman. „An MDP-based recommender system“. In: *Journal of Machine Learning Research* 6.Sep (2005), pp. 1265–1295.
- [46] Bichen Shi, Makbule Gulcin Ozsoy, Neil Hurley, Barry Smyth, Elias Z Tragos, James Geraci, and Aonghus Lawlor. „PyRecGym: a reinforcement learning gym for recommender systems“. In: *Proceedings of the 13th ACM Conference on Recommender Systems*. ACM. 2019, pp. 491–495.
- [47] Richard S Sutton. „Learning to predict by the methods of temporal differences“. In: *Machine learning* 3.1 (1988), pp. 9–44.
- [48] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.
- [49] Nima Taghipour, Ahmad Kardan, and Saeed Shiry Ghidary. „Usage-based web recommendations: a reinforcement learning approach“. In: *Proceedings of the 2007 ACM conference on Recommender systems*. ACM. 2007, pp. 113–120.
- [50] Hado Van Hasselt, Arthur Guez, and David Silver. „Deep reinforcement learning with double q-learning“. In: *Thirtieth AAAI conference on artificial intelligence*. 2016.

-
- [51] Petar Velickovic. *TikZ / Multilayer perceptron (MLP)*. 2016. URL: <https://github.com/PetarV-/TikZ/tree/master/Multilayer%20perceptron> (visited on 09/09/2019).
- [52] Huazheng Wang, Qingyun Wu, and Hongning Wang. „Factorization bandits for interactive recommendation“. In: *Thirty-First AAAI Conference on Artificial Intelligence*. 2017.
- [53] Jun Wang, Arjen P De Vries, and Marcel JT Reinders. „Unifying user-based and item-based collaborative filtering approaches by similarity fusion“. In: *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM. 2006, pp. 501–508.
- [54] Xinxi Wang, Yi Wang, David Hsu, and Ye Wang. „Exploration in interactive personalized music recommendation: a reinforcement learning approach“. In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 11.1 (2014), p. 7.
- [55] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. „Duelling network architectures for deep reinforcement learning“. In: *arXiv preprint arXiv:1511.06581* (2015).
- [56] Christopher JCH Watkins and Peter Dayan. „Q-learning“. In: *Machine learning* 8.3-4 (1992), pp. 279–292.
- [57] Chunqiu Zeng, Qing Wang, Shekoofeh Mokhtari, and Tao Li. „Online context-aware recommendation with time varying multi-armed bandit“. In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2016, pp. 2025–2034.
- [58] Xiangyu Zhao, Liang Zhang, Zhuoye Ding, Dawei Yin, Yihong Zhao, and Jiliang Tang. „Deep reinforcement learning for list-wise recommendations“. In: *arXiv preprint arXiv:1801.00209* (2017).
- [59] Xiaoxue Zhao, Weinan Zhang, and Jun Wang. „Interactive collaborative filtering“. In: *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM. 2013, pp. 1411–1420.
- [60] Guanjie Zheng, Fuzheng Zhang, Zihan Zheng, Yang Xiang, Nicholas Jing Yuan, Xing Xie, and Zhenhui Li. „DRN: A deep reinforcement learning framework for news recommendation“. In: *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2018, pp. 167–176.