Reinforcement Learning for Addressing the Cold-User Problem in Recommender Systems

Stelios Giannikis[a], Flavius Frasincar[a,*], David Boekestijn[a]

[a] *Econometric Institute, Erasmus University Rotterdam, P.O. Box 1738, NL-3000 DR Rotterdam, the Netherlands*

Abstract

Recommender systems are widely used in webshops because of their ability to provide users with personalized recommendations. However, the cold-user problem (i.e., recommending items to new users) is an important issue many webshops face. With the recent General Data Protection Regulation in Europe, the use of additional user information such as demographics is not possible without the user's explicit consent. Several techniques have been proposed to solve the cold-user problem. Many of these techniques utilize Active Learning (AL) methods, which let cold users rate items to provide better recommendations for them. In this research, we propose two novel approaches that combine reinforcement learning with AL to elicit the users' preferences and provide them with personalized recommendations. We compare reinforcement learning approaches that are either AL-based or item-based, where the latter predicts users' ratings of an item by using their ratings of similar items. Differently than many of the existing approaches, this comparison is made based on implicit user information. Using a large real-world dataset, we show that the item-based strategy is more accurate than the AL-based strategy as well as several existing AL strategies.

Keywords: recommender systems, reinforcement learning, active learning, cold-user problem

1. Introduction

As many companies have turned from their usual market places to the online market and at the same time, customers are turning to online retailers for activities like shopping (Amazon, eBay) or watching movies (Netflix, Hulu), recommender systems are becoming increasingly important. Moreover, as the variety of available products increases in these webshops, customers face a choice overload [1]. As previous works have explained, this occurs when a customer has to choose amongst many options, which sometimes can lead to a decrease in choice satisfaction [2, 3]. However, providing personalized recommendations to new users (also known as cold users: users on which the system has no available information) is difficult as there is no information about these users. In addition, the recent General Data Protection Regulation in Europe[1]

*Corresponding author; tel: +31 (0)10 408 1340; fax: +31 (0)10 408 9162
Email addresses: steliosgiannik@gmail.com (Stelios Giannikis), frasincar@ese.eur.nl (Flavius Frasincar), boekestijn@ese.eur.nl (David Boekestijn)
[1] https://gdpr-info.eu/

makes the use of additional demographical [4] or social [5] information difficult without explicit consent from the user.

Furthermore, most researches are based on explicit information about the users, which usually includes ratings about products that the users provide. On the other hand, implicit information (e.g., the website navigation or the past purchase behaviour of the customer), is abundant and can be leveraged by companies to address the cold-user problem [6, 7].

With the rise of deep learning and Reinforcement Learning (RL), new and more sophisticated algorithms arise, which can be applied in several fields like robotics and computer gaming [8, 9], and can deal with complex problems [10]. RL techniques have been used recently in several researches that aim to provide personalized recommendations to the users [11, 12]; however, they have rarely been used to solve the cold-user problem, apart from some exceptions like [13], which apply RL using explicit user information.

In the field of recommender systems, Active Learning (AL) proposes items to the cold users to rate in order to provide them with recommendations [14]. RL can be combined with AL to improve and personalize those recommendations. Moreover, as implicit information is more frequent in webshops, this research will make use of implicit information about the users to extract their preferences using state-of-the-art techniques from the field of RL.

Besides using implicit feedback data from the users, four recent AL strategies proposed by [15] are used next to the strategies introduced by [16] as benchmarks. To address the cold-user problem, we use a Deep Q-learning Network [8], which is one of the most popular RL methods that shows promising results, and is able to effectively elicit the preferences of the cold users and provide them with personalized recommendations [17].

The main contributions of the paper are as follows. First, we explore two RL approaches to address the cold-user item recommendation problem: one item-based and one AL-based. To our knowledge, the combination of these approaches with *implicit* user data, which allows our approaches to comply with the General Data Protection Regulation, has never been explored before. Second, the proposed approaches have been validated on a large, publicly available real-world dataset from a popular store in the Netherlands. The code of our implementation is written in Python and made publicly available at `https://github.com/SteliosGian/ReLeCUR`.

The remainder of this paper is structured as follows. In Section 2, we discuss related work on recommender systems and (RL approaches for solving) the cold-user problem. We follow by describing the proposed methods in Section 3 and explain how these are evaluated in Section 4. Next, in Section 5, we provide and discuss the results obtained. Last, we conclude by providing a summary of our work and suggesting future research in Section 6.

2. Related Work

There are approaches in the literature related to RL techniques in recommender systems both for making recommendations [11, 12] and for handling the cold-user problem [13]. There are several methods for providing recommendations to the users and amongst them are the collaborative filtering and content-based filtering methods, although a combination of these two is used in most situations and this combination is included in a category of recommendation systems called hybrid recommendation systems [18]. The content-based filtering method finds similar items from those that the user has already bought based on item descriptions and other characteristics. By using the item similarity, it is able to make recommendations to the user and also to recommend cold items (i.e., items that have not been purchased yet by a user). On the other hand, collaborative filtering searches for similar users to a targeted user and recommends items based on what similar users have liked.

In what follows, we will mainly focus on the collaborative filtering method as it outperforms other methods in most situations [19]. Several approaches in collaborative filtering exploit additional information about the users to improve recommendations [20]. Of these methods, AL has shown to be able to deal with the cold-start problem, because it obtains more (high-quality) data better representing the users' preferences [20]. In our approaches, we use implicit data to best represent the users' preferences as well as a different architecture for the Deep Q-learning Network compared to [13]. We will first discuss the collaborative filtering and matrix factorization methods, which are at the core of employed recommender systems. Later, we focus more on the cold-user problem, and AL and RL approaches for solving this problem.

2.1. Collaborative Filtering

Amazon.com, one of the largest webshops worldwide, relies heavily on recommendation systems to provide personalized recommendations to its numerous customers. Smith and Linden [21] indicate that in 1998, Amazon started using an item-based collaborative filtering for providing recommendations to its customers. Since then, this algorithm has been used by many different companies across the web. Item-based collaborative filtering was an improvement over its predecessor, the user-based collaborative filtering algorithm which was used in the mid-1990s [21]. The user-based algorithm finds users that have similar purchase or rating patterns to the targeted user, however, there are scalability issues as it becomes computationally expensive in large datasets. The item-based algorithm on the other hand, finds similar items to the item that needs to be predicted based on the already purchased or rated items of the targeted user. This algorithm is faster than its user-based counterpart, because there are usually more users than items in a webshop, and can scale to many items and users [22]. Furthermore, Smith and Linden [21] note the importance of time in the recommendations. As indicated, users get old and change habits over time and some items are bought before others. For instance, a memory card for a camera should not be recommended before a user buys a camera. Moreover, diversity in recommendations is also an important aspect [1, 21] as recommending items previously unseen by the user can sometimes bring more benefits than recommending always similar items.

Although explicit feedback data such as ratings have been used extensively in the research, they are hard to acquire compared to implicit feedback data. The user-item interaction matrix in explicit feedback datasets is usually extremely sparse, leading to problems in providing recommendations. Implicit feedback on the other hand, includes indirect information about the preferences of the users such as past purchases.

As Hu et al. [23] mention, implicit feedback does not indicate a negative opinion of the user about an item. In explicit feedback (e.g., ratings), users can provide low ratings to an item to express their negative opinion; however, this is not the case in implicit feedback as not buying an item does not necessarily mean that the user does not like that item. It might be the case that the user does not know about the item or she has no opinion about it. In addition, buying an item can also not mean a strong liking in that item as there is a possibility for that item to have been purchased as a gift, or for the user to have been disappointed after the purchase and returned the product.

As Hu et al. [23] also note, implicit feedback systems need to take into account other factors such as the availability of items. Due to the lack of the sparsity problem in these systems, as implicit feedback is available for all the users in the system, the user-item interaction matrix can become extremely large, therefore making optimization techniques such as stochastic gradient descent difficult to use without more computational power. For that reason, the alternating-least-squares technique is preferred over stochastic gradient descent in these cases [24].

Apart from single methods for recommendation systems, hybrid methods also exist that combine two or more methods together [25]. An extensive review of hybrid recommender systems has been made by Burke [26]. Burke presents several methods to combine different models into a hybrid model, and amongst the best ones are the *Cascade* and the *Feature Augmentation* methods. The *Cascade* method contains two recommendation systems where one refines the output of the second, and the *Feature Augmentation* is where the features that are created by one recommendation system are fed into the second recommendation system which gives the final recommendations. An example that is given by [26] is when the classifier of a content-based filtering model creates additional ratings for the users, therefore making the user-item matrix less sparse. In general, hybrid models usually perform better than single ones as each model covers the disadvantages of the other. A noticeable example of a hybrid system is the one which combines the collaborative with the content-based filtering, where the content-based model is able to address the cold-item problem that the collaborative model cannot, and the collaborative model can provide superior predictions compared to the content-based.

A different method involves two-layer undirected graphical models which are called *Restricted Boltzmann Machines* and was proposed by Salakhutdinov et al. [27]. Restricted Boltzmann Machines outperform models using Singular Value Decomposition. However, more advanced models that come from the field of deep learning have emerged. Wang et al [19] proposed a hierarchical Bayesian model called *Collaborative Deep Learning* (CDL). CDL is a model that tightly couples (i.e., allows two-way interaction) the deep representation learning for the content information and the collaborative filtering for the ratings matrix. Moreover, CDL is a general framework that can incorporate other deep learning models such as Deep Boltzmann Machines, recurrent neural networks, and convolutional neural networks. CDL seems to outperform all the other models considered, however, it lacks explainable results as it is based on deep learning methods [19].

2.2. Matrix Factorization

The idea of representing the users and items with latent factors has been initially proposed by Funk [28]. However, this technique became famous through the Netflix prize competition by Koren et al. [29] as it succeeded in reaching the first place. *Matrix Factorization* models create latent factors that represent the users and the items. One advantage of these kind of models is that they can incorporate extra information. This can be more evident in the case of implicit data where there are many variables indicating the user behaviour like the user's past purchases or the user's website movement behaviour. Furthermore, users that are constantly rating items high or the presence of extremely popular items can hinder the analysis and provide misleading results. For that reason, biases can be added in matrix factorization models to remedy this problem. Moreover, temporal dynamics can be included into the matrix factorization models which improves accuracy as they can capture the changes in the popularity of items (items lose their popularity over time) or in the behaviour of users.

Amongst the different types of the matrix factorization method, there is also the Regularized Singular Value Decomposition [30] that includes biases to the Singular Value Decomposition algorithm and uses the kernel ridge regression for the post-processing of the results. According to Paterek [30], the results were 7.04% better than of Netflix's "Cinematch" algorithm. Different variations of the matrix factorization algorithm have been proposed like Bayesian probabilistic matrix factorization that uses a Markov chain Monte Carlo [31], non-negative matrix factorization [32] and others.

2.3. Cold-User Problem

The cold-user problem appears to be one of the most serious problems for webshops even today [33]. Providing the new users with personalized recommendations is significant as it increases the customer base of a company and, consequently, its market share. In addition, new webshops need to maintain their customer base since it is already small and losing customers will lead to a decrease in revenues and market share [34]. For that reason, several approaches have been proposed for addressing the cold-user problem.

An approach towards solving the cold-user problem has been proposed by [35]. This method, which is called *Functional Matrix Factorization* (fMF), makes use of AL. In this case, users are presented with some items when they enter the webshop in order to express their preferences. The fMF method tries to combine the traditional matrix factorization algorithm with a decision tree, where each node of the decision tree represents an interview question. Moreover, as the user progresses through the interview, the next interview question takes into account the answer of the previous one. This is considered a personalized AL approach according to [14]. This method performs well in the cold-user scenario whereas in the warm-user (i.e., users that have interacted with the system) scenario it seems to perform worse than the traditional matrix factorization algorithm.

Another interesting approach, which is based on implicit feedback data is presented by [16]. In [16], the authors use traditional AL methods for recommender systems [14], but adjusting them for implicit feedback.

In addition, they suggest two new AL strategies called Gini and PopGini where the latter is a combination of the Popularity and the Gini strategies. The PopGini strategy seems to outperform all the other AL strategies except when the number of shown items k is 10 where the random strategy is superior. In the scenario with 25, 50, and 100 shown items, the PopGini strategy is superior to the other AL strategies which are the Random, Popularity, Gini, Entropy, and PopEnt (i.e., Popularity and Entropy) strategy. Further, four additional AL strategies (i.e., Misclassification Error, PopError, Variance, and PopVariance) were proposed by [15]. An important note is that these are all non-personalized AL strategies according to [14], which means that the previous answers of the users are not taken into account.

2.4. Reinforcement Learning

RL is considered a part of the machine learning field like supervised and unsupervised learning. Although RL is mainly used in the field of robotics [9] and games [36], it has also been applied to the field of recommender systems [11, 12, 37, 38], but rarely in situations where the cold-user problem is present [13, 39]. RL involves an agent who takes actions in an environment with the goal of maximizing a cumulative reward [40].

Several approaches have been made in the field of recommender systems both for providing recommendations to the warm users and to the cold users. An interesting approach that is focused on the field of online learning platforms (Coursera, Udacity, etc.), is presented by [37]. In their research, the authors developed a recommender system by using RL for online learning websites like Khan Academy and Coursera which need to provide personalized recommendations to their users. The RL model uses Q-learning in which an optimal policy π_* has to be found. However, in this approach, the problem of cold users is not addressed.

One of the main problems in using RL in the field of recommender systems, is the large action space that exists. In general, the actions are what the agent can do (e.g., in a game environment, actions can be to move right or left). However, in recommender systems, the actions are usually the items (i.e., which item to recommend to the user). Due to the numerous available items that can be recommended, the action space becomes too large. The authors of [41] propose a method to provide personalized recommendations through the use of a biclustering technique that reduces the action and state space of the agent and improves the quality of the recommendations. In this case, the state space is an $n \times n$ grid world with n^2 states where n^2 is the number of biclusters which is defined arbitrarily (n for users and n for items) and each state is a set of users and items that were derived from the biclustering technique. The action space is a set of four actions (right, left, up, and down). The algorithms that were used were Q-learning [42], an off-policy method, and SARSA [43], an on-policy method. Off-policy means that the agent learns the optimal policy by exploring new policies and using previous experience, while on-policy means that the agent does not explore new policies.

One method for addressing the cold-user problem with RL is presented by [13]. In this paper, an AL approach is used to provide items to the cold users to rate in order to elicit their preferences and provide better recommendations. For that, an interview is created, but the items that will be presented to the user will be selected using RL. In this case, the Q-learning algorithm was used to create the interview, and more specifically, Deep Q-learning Network [8], which uses a deep neural network to approximate the optimal Q^* function. As mentioned before, the main problem in the field of recommender systems with RL is the large action space. To address this issue, the researchers use only the 100 most frequently rated movies as possible recommendations to the cold users. Their neural network is a multilayer perceptron with two hidden layers where the first layer contains 64 neurons while the second layer contains 32 neurons, and both have a tanh activation function. Last, the output layer has 100 neurons and a ReLU activation function. The recommender system was tested on the MovieLens dataset. This method is compared with other famous methods like Tree [44], TreeU [44], fMF [35], and Q-Embedding, and it outperforms them by using 3 or 4 interview questions. As mentioned, most of the literature focuses on recommender systems using explicit data, while no research, to the best of our knowledge, utilizes implicit data employing RL techniques to address the cold-user problem.

3. Methodology

In this section, the dataset that is used for this research will be presented, the two distinct RL approaches, as well as the four additional AL strategies proposed by [15], namely the Error, PopError, Variance, and PopVar, which are added to the existing set of strategies that were tested in an implicit feedback dataset provided by the authors of [16]. The two RL approaches are the item-based approach, where the actions of the agent are the items, and the AL-based approach, where the actions of the agent are the AL strategies. To enhance the reader's understanding, we provide a high-level overview (Fig. 1) of the Deep Q-Learning and matrix factorization models' architectures and interaction.

3.1. Dataset

The dataset that will be used for this research is an implicit feedback dataset from a Dutch department store named *De Bijenkorf*[2]. De Bijenkorf has a large webshop that attracts over 100,000 visitors per day and has over 200,000 products. The dataset contains three variables, which are the user id, the item id, and an interaction variable that shows whether there was an interaction between a user and an item. To create this variable, if a user has purchased an item, the interaction value for that user and item (r_{ui}) is 1 whereas if the user has returned that item, the interaction value is 0. The same dataset is used for a similar research conducted by [16] and was taken from the personal GitHub page of one of the authors[3].

The dataset contains interactions that have been done between the 14th of July 2015 and the 13th of July 2016. It contains 563,495 unique customers and 242,020 unique products. In total, the number of unique interactions that are available in the dataset are 2,563,878. Out of all the interactions, 2,159,538 of them are positive (i.e., have an interaction value of 1), and 404,340 are negative (i.e., have an interaction value of 0).

[2]`https://www.debijenkorf.nl/`

[3]https://github.com/Tomas92/addressing_the_cold_user_problem_for_model-based_recommender_systems

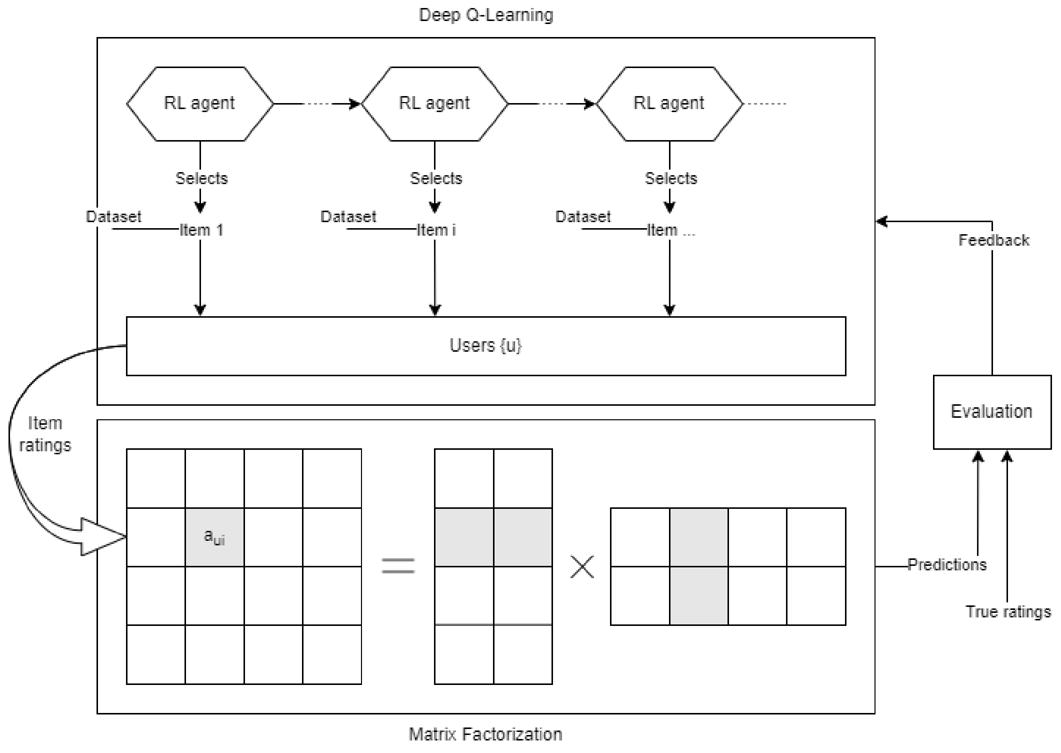

Figure 1: High-level overview of the models' architectures and interaction. Note that the selection process ('Selects') is an involved process and depends, among others, on the RL approach being item-based or AL-based.

On average, a customer of the webshop has interacted with 4.55 items and every item has been interacted with by 10.59 customers.

3.2. Active Learning Strategies

AL, in the case of the cold-user problem in recommender systems, helps find the best items to give to the user to rate in order to minimize an overall loss function. There are several strategies that select specific items from the item pool to be rated by the users to elicit their preferences and some of them are provided by [14] while two new strategies are provided by [16].

These strategies can be split between *personalized* and *non-personalized.* When using a personalized AL strategy, the first item that a cold user rates influences the selection of the second item that the system will provide to the user to rate, whereas in a non-personalized strategy, this is not the case. In addition, each strategy can be either *single-heuristic* or *combined-heuristic.* Single-heuristic strategies apply one rule to select the best set of items to be given to the user to rate while combined-heuristic combine two or more single-heuristic strategies.

Next, the non-personalized single-heuristic strategies can be split between three types: the *uncertainty reduction*, the *error reduction*, and the *attention-based.*

Strategies that are of the uncertainty-reduction type, aim at finding and providing items to the cold user that have very diverse ratings, meaning not popular or very unpopular items. The ratings of these items by the cold user can provide useful information about the user's preferences. Furthermore, the error-reduction strategies try to increase the predictive accuracy of the model by providing items to the cold user that will reduce the error (e.g., Root Mean Squared Error). Finally, attention-based strategies select items that have high attention from the users. One such strategy is the popularity strategy which selects the most popular items to be given to the cold users to rate. In the following sections, these AL strategies will be presented and explained.

3.2.1. Popularity Strategy

The first strategy that will be used is the Popularity strategy. The popularity strategy is considered an attention-based strategy that provides to the user a number of the most popular items to rate and the number of given items is defined by the system. This means that items are ranked based on their popularity in the webshop (i.e., how frequently they have been purchased or rated). This strategy can be considered a simple strategy, compared to other more advanced strategies that will be presented in the following sections. This strategy is important, as it can show items that the users are most interested in, because these items have been purchased or rated many times compared to other items.

3.2.2. Entropy Strategy

The Entropy strategy is considered as a non-personalized single-heuristic strategy which belongs to the uncertainty-reduction group of strategies.

The entropy strategy checks an item and measures how scattered its ratings are. In the case of De Bijenkorf, the ratings are considered binary (purchased or returned) and for that reason, the Shannon's entropy will be used:

$$Entropy(i) = -\sum_{j \in 0,1} p(j|i) log_2 p(j|i). \tag{1}$$

In this equation, $p(j|i)$ is the relative probability that a user has purchased the item i (j = 1) or has returned it (j = 0). A high entropy means that the ratings (or the interactions) for this item are dispersed.

3.2.3. PopEnt Strategy

Next, a combined-heuristic strategy is the PopEnt Strategy that combines the popularity score with the entropy strategy. However, due to the fact that the distribution of the two strategies differ, the logarithm with base 10 is taken for the popularity score. That is because the distribution of the popular items is exponential while in the case of entropy, it resembles more the normal distribution. If the logarithm was not taken, then the popularity score would have higher weight than the entropy score. To acquire the PopEnt score for each item, a weighted sum of the two methods was taken, where the weights for each strategy were configured from a range of possible weights. The PopEnt strategy is given by the following equation:

$$PopEnt(i) = w_1 \cdot logPopularity(i) + w_2 \cdot Entropy(i), \tag{2}$$

where w_1 is the weight of the Popularity strategy and w_2 is the weight of the Entropy strategy.

3.2.4. Gini Strategy

The Gini as well as the PopGini strategy was introduced by [16] as an additional AL method that showed promising results. The Gini impurity measure tries to split the items and provide the ones that have diverse ratings, which supposedly will give better information about the users' preferences. Like the Entropy strategy, Gini is considered a non-personalized single-heuristic strategy of the uncertainty-reduction group of strategies. The Gini measure is given by the following equation:

$$Gini(i) = 1 - \sum_{j \in 0,1} (p(j|i))^2. \tag{3}$$

In (3), the $p(j|i)$ is the relative frequency of a positive or a negative interaction for the item i where j = 1 refers to a positive interaction (i.e., the user has bought the item i) and j = 0 refers to a negative interaction (i.e., the user has returned the item i).

3.2.5. PopGini Strategy

The PopGini strategy is a weighted sum of the Popularity and the Gini strategy with the same weights as the PopEnt strategy. It is considered a non-personalized combined-heuristic ranking strategy with a combination of attention-based (Popularity) and uncertainty-reduction based (Gini) strategies. The weighted sum is calculated from the logarithm of the scores of the Popularity strategy and the scores of the Gini strategy as follows:

$$PopGini(i) = w_1 \cdot logPopularity(i) + w_2 \cdot Gini(i). \tag{4}$$

3.2.6. Error Strategy

The Error strategy is a static non-personalized single-heuristic strategy used for the item ranking that aims at finding items that reduce the misclassification error. The Error strategy for an item i is given by:

$$Error(i) = 1 - \max(p(j|i)), \tag{5}$$

where $\max(p(j|i))$ is the maximum relative frequency of a positive (j = 1) or a negative (j = 0) interaction for an item i.

3.2.7. PopError Strategy

The PopError strategy is another strategy for addressing the cold-user problem, which is a combination of the Popularity and the Error strategies that tries to exploit the advantages of both the Popularity and the Error strategies. For this reason, a weighted sum of both strategies is being considered with weights similar to the other combined-heuristic strategies.

$$PopError(i) = w_1 \cdot logPopularity(i) + w_2 \cdot Error(i). \tag{6}$$

3.2.8. Variance Strategy

Next, the Variance strategy aims at finding items with the largest variance in their ratings (interactions). Items with high score on variance are considered uncertain, as the system does not have clear ratings for those items. The variance strategy is also considered a non-personalized single-heuristic strategy as it utilizes only one method (Variance) for the ranking of the items. The equation for the Variance strategy is given by:

$$Variance(i) = \frac{1}{|U_i|} \sum_{u \in U_i} (r_{ui} - r_i)^2, \tag{7}$$

where $|U_i|$ is the number of users u who have interacted with the item i, r_{ui} is the rating (either 1 or 0) for the item i. Finally, r_i is the mean rating for the item i, taking all the users who have interacted with the item i either positively or negatively.

3.2.9. PopVar Strategy

The PopVar strategy is a combination of the Popularity and the Variance strategies, thus making this a non-personalized combined-heuristic strategy. Similarly to the other combined-heuristic strategies, the weighted sum of the Popularity and the Variance strategies is being taken. The PopVar strategy is given by:

$$PopVar(i) = w_1 \cdot logPopularity(i) + w_2 \cdot Variance(i). \tag{8}$$

3.3. Deep Reinforcement Learning

RL is considered a sub-field of machine learning, like supervised and unsupervised learning. However, as opposed to these two classical areas of machine learning that aim at predicting a dependent variable in the case of supervised learning, or finding some underlying patterns in the case of unsupervised learning, RL uses methods of "trial and error" to achieve its goal.

3.3.1. Markov Decision Process

RL involves an agent that interacts with an environment by taking certain actions. With this interaction, the agent gets a reward, which is a numerical value, and the ultimate goal of the agent is to maximize its cumulative reward. In this section, the RL method will be explained and the notation used is taken from [40].

To start, an RL problem can be mathematically formulated as a Markov Decision Process (MDP) which is defined by (S, A, R, P, γ) where:

- S: A finite set of possible states,
- A: A finite set of possible actions,
- R: The distribution of the reward given a state-action pair,
- P: The state transition probability matrix,
- γ: Discount factor $\gamma \in [0, 1]$.

The state transition probability matrix is defined as:

$$P_{ss'}^{a} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a], \tag{9}$$

which means that the transition probability matrix from a state s to a state s' depends on the taken action. Here, S_t and A_t, as well as R_{t+1} that will be introduced later, are random variables at time t and can take values s, a, and r, respectively. Next, the reward function is defined as:

$$R_s^a = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]. \tag{10}$$

The goal in a MDP is to find the best path that, through the decision-making process, maximizes the sum of the rewards (R_{t+1} is used instead of R_t to denote that the reward comes after an action is taken, although R_t is also found in the literature). To find the best path, there is a stochastic policy π that is a distribution over actions given states, which is defined as:

$$\pi(a|s) = \mathbb{P}[A_t = a | S_t = s]. \tag{11}$$

Some essential terms are the state-value function and the action-value function which will be explained next. The state-value function $v_\pi(s)$ when following a specific policy π and being in a state s, shows what is the total reward from that state onward. In other words, if an agent begins from a particular state s, the state-value function shows what is the total reward from that state until the MDP is terminated. The state-value function is defined as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s], \tag{12}$$

where G_t is the cumulative reward $G_t = \sum\limits_{\kappa=0}^{\infty} \gamma^\kappa R_{t+\kappa+1}$.

Similar to the state-value function, the action-value function $q_\pi(s, a)$ shows the quality of a particular action a from a particular state s while following a policy π and is defined as:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a], \tag{13}$$

which is the expected return G_t of taking an action a from a state s while following a policy π.

Next, both the state-value and the action-value functions can be split into the immediate reward from the initial state and the discounted value of the next state. This is the Bellman Expectation equation for the state-value and the action-value functions:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma\, G_{t+1} \,|\, S_t = s] \\ &= \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P^a_{ss'}[R^a_s + \gamma\, \mathbb{E}_\pi[G_{t+1} \,|\, S_{t+1} = s']] \\ &= \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P^a_{ss'}[R^a_s + \gamma\, v_\pi(s')], \end{aligned} \tag{14}$$

$$\tag{15}$$

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma\, G_{t+1} \,|\, S_t = s, A_t = a] \\ &= \sum_{s' \in S} P^a_{ss'}[R^a_s + \gamma \sum_{a' \in A} \pi(s',a') q_\pi(s',a')]. \end{aligned} \tag{16}$$

To solve the Markov Decision Process, the optimal state-value and action-value functions are needed to be found. In details, the optimal state-value function $v_*(s)$ is the maximum state-value function over all policies π and the optimal action-value function $q_*(s,a)$ is the maximum action-value function over all policies π. These can be expressed as:

$$v_*(s) = \max_\pi v_\pi(s), \tag{17}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a). \tag{18}$$

More specifically, $v_*(s)$ shows the maximum reward an agent can receive from a state s, and $q_*(s,a)$ shows the maximum reward an agent can receive after taking an action a, starting from a state s. In general, the action-value appears to be the most important, as knowing what is the best action to take from a particular state, can solve the MDP problem and the agent can behave optimally.

Finding the optimal policy π_* in a RL problem seems to be a crucial part. In order for a policy π to be optimal, it has to be better than every other policy, thus, the value-function of that policy needs to be better than or equal to the value-function of every other policy in every possible state. We denote a policy to be better than another policy as follows:

$$\pi \geq \pi' \text{ if } v_\pi(s) \geq v_{\pi'}(s), \forall s. \tag{19}$$

To find the optimal policy π_*, $q_*(s,a)$ needs to be maximized:

$$\pi_*(a|s) = \begin{cases} 1, & \text{if } a = \operatorname*{argmax}_{a \in A} q_*(s,a); \\ 0, & \text{otherwise.} \end{cases} \tag{20}$$

Finally, to find the optimal values, the Bellman Optimality equation is used:

$$\begin{aligned} v_*(s) &= \max_a q_*(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma\, G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma\, v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s' \in S} P^a_{ss'}[R^a_s + \gamma\, v_*(s')], \end{aligned} \tag{21}$$

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] \\ &= \sum_{s' \in S} P^a_{ss'}[R^a_s + \gamma \max_{a'} q_*(s', a')]. \end{aligned} \tag{22}$$

However, this is difficult to be solved in practice as these equations are non-linear due to their recursivity, and there is no closed-form solution. Many methods have been proposed to solve this issue and amongst them are the value iteration, policy iteration, Q-learning [42], and SARSA [43] methods. In the following section, the Q-learning method will be explained, which combined with deep learning, is one of the most popular and recent methods in the field of RL. For that reason, this will be the method that will be used to address the cold-user problem.

3.3.2. Q-learning

Q-learning [42], is a way to solve the issue of non-linearity of the Bellman Optimality equation and consequently, find the optimal policy π_*. So, to find π_*, the different policies are needed to be evaluated. Moreover, by evaluating the target policy $\pi(a|s)$, the computation of $v_\pi(s)$ or $q_\pi(s,a)$ is possible. Q-learning is considered an off-policy learning method in which the agent tries to learn the *target policy* $\pi(a|s)$ by following a *behavior policy* $\mu(a|s)$. (On-policy learning on the other hand, tries to evaluate the target policy π by sampling experience from the same target policy.) Off-policy methods can find the optimal policy by following an exploratory strategy where random actions from a set of actions are chosen, and by taking optimal actions based on previous experience (i.e., from previously followed policies). This is known as the explore/exploit dilemma where the agent needs to choose between following an exploration or an exploitation policy at each step. Moreover, the target policy will eventually become the optimal policy π_*.

In Q-learning, the goal is to learn the action values $Q(s,a)$. This is done by choosing an action while the agent is in a state, using the behavior policy $A_t \sim \mu(a|S_t)$. Next, an alternative action is considered which was not taken $A' \sim \pi(a|S_t)$. Lastly, the $Q(S_t, A_t)$ is updated by taking into consideration the value of the alternative action and having a learning rate α:

$$Q(S_t, A_t) \leftarrow (1-a)Q(S_t, A_t) + \alpha\big(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a')\big). \tag{23}$$

As mentioned, Q-learning follows either an exploration strategy or an exploitation strategy for each action to find the optimal policy. Moreover, the target policy is a greedy policy with respect to $Q(s, a)$ and is learned by following an exploration policy:

$$\pi(S_t) = \underset{a'}{\operatorname{argmax}}\, Q(S_t, a'). \tag{24}$$

Furthermore, the behavior policy is ϵ-greedy with respect to $Q(s, a)$, which means that it needs to choose between following an exploration strategy or a greedy strategy at each step. However, in order for large problems to be solved using RL, there needs to be some scaling methods as many problems include a large state space. For instance, some examples of these problems are the Backgammon game which has 10^{20} states and the Computer Go game that has 10^{170} states. As large Markov Decision Process problems involve a large state space, these are usually difficult to be stored in memory and to compute the values of the states. A common solution to this problem is the use of a parametric function approximator that tries to approximate either the state-value or the action-value function where $\mathbf{w}$ are the weights. By using a function approximator, it is possible to generalize also from observed to unobserved states.

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s), \tag{25}$$

$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a). \tag{26}$$

Common function approximators are the Neural Networks and especially Deep Neural Networks. In the case of Q-learning, the use of a deep neural network as a function approximator is called a Deep Q-learning Network (DQN) [8]. DQN uses the method of experience replay to store the transitions and sample random transitions from the stored ones. By using these transitions, the weights are updated by using stochastic gradient descent that will help approximate the action-value or the state-value function.

In DQN, the actions are selected based on a ϵ-greedy policy. In addition, the transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ are stored in the replay memory D to be used later. Next, some of the transitions are sampled (s, a, r, s') using a minibatch and the Q-learning targets are computed with respect to the fixed parameters w^-. These fixed parameters w^- are the weights of the neural network that are kept fixed for a specific number of steps C. One main problem that the model encounters, is that it can become unstable if the target values (the fixed parameters) change continuously as the model becomes better. This is solved by keeping the target values fixed for some updates (w^-) and then, the target values can be updated (w). Finally, the loss function is being minimized which is the squared difference between the prediction of the Q-network and the target Q-network.

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim D}\left[\left(r_{t+1} + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i)\right)^2\right]. \tag{27}$$

The above equation shows the loss function, where w_i^- are the fixed weights for C iterations, and w_i are the weights at iteration i.

The reason experience replay is used in DQN, is because it helps reduce the correlation between the paths. For instance, if the last available transitions were taken, then these transitions would be highly correlated as the agent might have followed a specific path. This method contributes to the stabilization of the neural network approximator.

In the case of the recommender system, there are two strategies that can be followed to solve the cold-user problem. One strategy is to consider the actions in the DQN as the items that will be suggested to the user. In this strategy, the model will try to find the best items to be rated by the users, which will reduce the Root Mean Squared Error (RMSE). The second strategy involves the AL strategies [14, 16] that can be used as actions instead of the items. These AL strategies like the PopEnt or the Entropy strategy, try to find the best items to be given to the cold users in order to rate them, therefore eliciting their preferences for subsequent recommendations. In addition, an improvement over Experience Replay is the Prioritized Experience Replay [45], which can be utilized to increase the performance of the agent and improve the stability of the model. According to [45], the idea behind Prioritized Experience Replay is to sample observations that might be more important than other observations but rarely occur. Sampling observations in a uniformly random manner from those transitions, may omit some observations that are useful in the training of the model (the ones with the largest error). To include the Prioritized Experience Replay to the DQN model, priorities p are given to each transition, which are defined as:

$$p = |\delta| + e, \tag{28}$$

where $\delta = r_{t+1} + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i)$ is the loss function represented in (27). The small positive value e is added to avoid cases where the error is 0. This is done because, if e was not added, then the priority for this transition would be 0, even though this transition could be useful for the model. Next, to sample the transitions, the priorities p are converted into probability values as shown in (29).

$$P(i) = \frac{p_i^a}{\sum_k p_k^a}. \tag{29}$$

The above equation shows the probability of choosing a specific transition from the current set of transitions. However, to avoid having some priority values too high and others too low, the a value is being added, which ranges from 0 to 1. An a value of 1, indicates that full priority sampling is being done, and an a value of 0, means that the normal uniform random sampling takes place.

With this process, the transitions are sampled based on the priority sampling technique, however, since the transitions are now sampled non-uniformly, this creates a bias in the neural network towards transitions that have high priority. To deal with this issue, the weights and biases of the neural network are being updated and scaled down to avoid overfitting. Overfitting can occur, because the neural network might learn the priority values, as the distribution of the transitions is changed in favour of the prioritized transitions. For this reason, a weight is being added to address this issue. This weight is given by:

$$weight_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^b, \tag{30}$$

where $weight_i$ is the importance sampling weight, N is the number of transitions that are taken for the training procedure, and $P(i)$ is the sampling probability. Because the bias correction is more important

ALGORITHM 1: Deep Q-Learning Algorithm

```
Initialize replay memory D with capacity N
Initialize Q(s,a) randomly
for episode ∈ episodes do
    while s is not terminal do
        With probability $\epsilon$ choose a random action a ∈ A,
        otherwise choose a = $\max_a\ Q(s,a;w)$
        Take action a and observe r, $s'$
        Save the transition $(s,a,r,s')$ in D
        Sample a minibatch of transitions $(s_j,a_j,r_j,s'_j)$ with Prioritized Experience Replay from D
        for each transition in minibatch do
            Set $y_j \leftarrow \begin{cases} r_j, & \text{for } terminal\ s'_j \\ r_j + \gamma \max_a Q(s'_j,a'_j;w^-), & \text{for } non-terminal\ s'_j \end{cases}$
            Apply gradient descent step on the loss
            $\left(y_j - Q(s_j,a_j;w)\right)^2$
            $s_j \leftarrow s'_j$
        end
        Every step, reduce the $\epsilon$ by a constant value $\epsilon$-decay ($\epsilon \leftarrow \epsilon - \epsilon$-decay)
        Every C steps, the fixed network is updated $w^- \leftarrow w$
    end
end
```

later in the training procedure, a value b is added, which starts from 0 in the early stages of training and arrives to 1. The $weight_i$ value adjusts the weights of the neural network to avoid overfitting because of the changed transition distribution, which is the result of the prioritized experience replay technique. This is done by multiplying the importance sampling weight by the weights (w_i) of the neural network.

In Algorithm 1, the DQN pseudo-code is being presented. At first, the capacity of the replay memory N is being defined and $Q(s,a)$ values are assigned randomly. For a pre-defined number of episodes, and while the agent has not reached the terminal state, a random action a is chosen with pre-defined probability ϵ, otherwise, an action is chosen based on past experience. The transitions are saved, and a minibatch of transitions is being taken with prioritized experience replay from the batch of transitions, to train the neural network. Next, y_j is set to either r_j, if the agent has reached the terminal state, or $r_j + \gamma \max_a Q(s'_j, a'_j; w^-)$, if the state is non-terminal, and the gradient step is applied on the loss. After pre-defined C steps, the target network is updated.

3.4. Model Creation

At first, the matrix factorization model needs to be created as the recommendations that will be made will utilize this method. The goal of the method is to minimize the RMSE, therefore providing personalized recommendations to the users. The number of items that will be shown to the users can vary, to assess the model in different situations.

To provide accurate recommendations, after the preferences of the users are elicited, the matrix factorization model needs to be tuned appropriately. The matrix factorization method splits the user-item interaction matrix into two matrices, one including the users and the latent features that are created, and the other including the items and the latent features. However, the number of latent features needs to be pre-defined. In addition, the learning rate of the stochastic gradient descent has to be defined, as well as the regularization term of the parameters in the matrix factorization model. These hyperparameters were found using the k-fold cross-validation method with $k = 10$. In the k-fold cross-validation method, the training set is split into k parts, where at each iteration, $k - 1$ parts are used as the training set and the remaining part is used as the test set. This method is commonly used in hyperparameter tuning to avoid overfitting where overfitting means that the model is not able to generalize in unseen data. In addition, the weights of the combined-heuristic AL strategies need to be defined using a range of possible values from 0 to 1.

Next, the number of cold users needs to be defined. This number was set to 25% of the total number of users which were randomly selected. Similar works have also randomly selected the number of cold users as there is not a clear rule for that selection [46]. For these users, the true interactions were hidden from the system and a number of items were shown to them based on the two RL methods that were created. If a user has not interacted with any of the shown items, then that user is excluded from the model as no recommendations can be made. Warm users as well as cold users [47] that interacted with at least one of the shown items were included in the training set, where the matrix factorization model will be trained upon. In the test set, where the model performance will be tested, the cold users were included that have interacted with at least one of the items presented to them. To sum up, the training set included all the warm users and the cold users with the shown items that they have interacted with while the test set included the cold users with their true items but excluding the ones that have already been shown to them.

The DQN model required an environment in order for the agent to operate and find the best items to be presented to the users. However, due to the two approaches that would be implemented, which will be explained in the following sections, two different environments were required, one for each approach. In these environments, the RL agent would select the best items and based on each situation's RMSE value, a reward would be given, which is the inverted RMSE value. The DQN model, in order to function optimally in this environment, has several hyperparameters that need to be set before the agent could start the training procedure. These hyperparameters, as were discussed in Section 3.3.2, are the γ value, which is the discount factor of future rewards, the learning rate of the Adam optimizer α [48], the ϵ value for the ϵ-greedy strategy (i.e., the exploration value), as well as the percentage by which the ϵ value will decrease (ϵ- decay). In addition, as the target network is not updated immediately, but after a specific number of steps C, this number of steps C needs to be optimized. Finally, for the training of the agent after it had gathered enough transitions, the number of those transitions (i.e., the Batch Size) needs to be determined as well as the total number of transitions (s, a, r, s') to be saved (i.e., the Buffer Size), from where the agent would select some of them for training. Finally, the neural network needs to be created as well as its architecture.

3.4.1. Item-based RL Approach

The first RL approach that utilized the Deep Q-learning technique to find the best possible actions for each state, is a model where the actions are defined as the items of the system. This first approach entails a serious problem regarding the action space, as it is large, making the algorithm difficult to converge to the optimal action for each state. A state is defined as the set of items that are presented to the cold user so far. The agent starts with an empty state and starts to fill it with possible items to be shown to the users. It then calculates the RMSE, which is used to define the reward system. To counter the main issue of this approach, a specific number of items needs to be defined that will be considered as actions, in order to reduce both the action space and the state space, and allow the algorithm to converge. The size of the state space is especially important, as training of the DQN model has a time complexity of $\mathcal{O}(n^3)$, with n the number of states [49]. The items can be taken from the Popularity strategy to consider a certain number of the most popular items as the actions and thus limit the action space and state space. A representation of the item-based RL approach can be seen in Fig. 2.

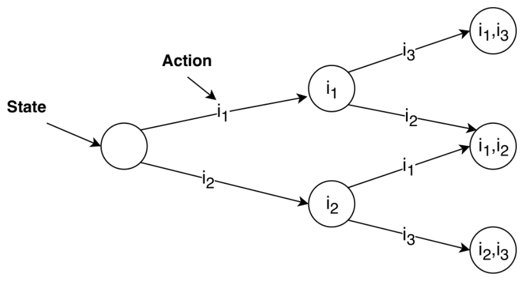

Figure 2: RL with Items as Actions.

3.4.2. AL-based RL Approach

The second RL approach that also makes use of the Deep Q-learning technique, defines the actions as the AL strategies. More specifically, 9 AL strategies, namely the Gini, PopGini, Entropy, PopEnt, Error, PopError, Variance, PopVar, and Popularity strategy were set as the 9 possible actions that the agent can take to move through the state space. Again, a state is defined like in the previous RL model, which is the items that are presented to the cold user so far. As before, one calculates the RMSE. However, the items in this approach are ranked based on each AL strategy, and then the agent selects one of the strategies, from where the first item in ranking is taken. In case where the agent selects a specific strategy more than once during an episode, the next item in ranking is taken from this strategy to avoid having the same item in a state. This approach shrinks the action space, thus dealing with the issue of the large action space of the previous approach. With this approach, the action space is greatly reduced to 9 possible actions, in comparison to the previous approach. A representation of the AL-based RL approach can be seen in Fig. 3.

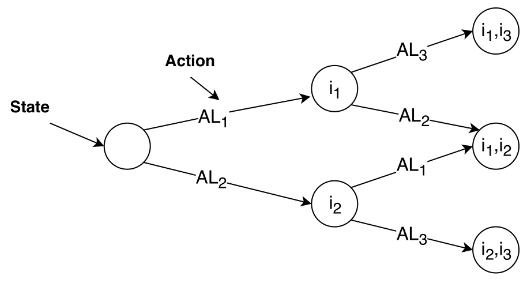

Figure 3: RL with AL Strategies as Actions.

4. Evaluation

As indicated, most of the literature is based on recommender systems using explicit feedback to understand the preferences of the users and provide them with personalized recommendations. In this research, the evaluation of the recommender system will be based on implicit feedback from the users. In this section, the evaluation metric used will be defined, as well as the model configuration and the parameters used to train the matrix factorization and RL models.

4.1. Evaluation Metric

In most approaches that are targeting recommender systems, the metric that is used to evaluate the performance of those systems is usually the RMSE. The RMSE metric can determine the quality of the recommendations, which shows if the predictions made for those users were correct. If the system has correctly predicted the interactions of the users, then the RMSE will be zero. The RMSE equation is given by:

$$RMSE = \sqrt{\frac{\sum_{(u,i)\in T}(a_{ui} - \hat{a}_{ui})^2}{|T|}}, \tag{31}$$

where a_{ui} is the true interaction of a user u for an item i and $\hat{a}_{ui}$ is the predicted interaction. The letter T stands for the test set, meaning the number of observations that are in the test set, with which the model has not been trained.

The RMSE metric is used for the matrix factorization model, but is also used for the RL model that will be created with some modifications. As already mentioned in the previous section, RL aims at maximizing its cumulative reward, which is defined by the system. For this case, the reward that the RL agent will try to maximize will be defined as:

$$Reward = \frac{1}{RMSE}, \tag{32}$$

where the RMSE is calculated from the matrix factorization model.

Finally, the loss function that the DQN is using is the *Huber Loss*, which combines the Mean Squared Error with the Mean Absolute Error. More specifically, it uses the Mean Squared Error for values less than or equal to 1, and the Mean Absolute Error for values greater than 1. The *Huber Loss* function is given by:

$$Huber(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq 1; \\ |a| - \frac{1}{2}, & \text{otherwise,} \end{cases} \tag{33}$$

where a is the difference between the observed value and the predicted value. The *Huber Loss* function has been used in the research that explained the DQN method [50].

4.2. Model Configuration

The number of items that were shown to the users were selected to cover a large range of shown items from 10 to 100 items. For this reason, the model was tested in situations with 10, 25, 50, and 100 shown items. In addition, for the item-based RL approach, the number of items that would be considered as actions were defined as the 200 most popular items. This step was taken to reduce the number of actions and allow the algorithm to converge. After defining the number of shown items and the actions of the item-based RL approach, the architecture of the neural network had to be defined. The Deep Q-learning Network consisted of a feedforward neural network with two hidden layers, consisting of 64 and 32 neurons respectively with a tanh activation function and the *Huber* loss function. These values were defined according to the research of [13]. The output layer consisted of 200 neurons in the case of the item-based RL approach and 9 in the case of the AL-based RL approach. The output layers were set according to the number of actions of each approach. For instance, since the item-based RL approach has 200 possible actions, the output layer has 200 neurons whereas the AL-based RL approach has 9 neurons as the number of possible actions is 9.

4.3. Model Parameters

The parameters to be used in both the matrix factorization and the Deep Q-learning models, are crucial to models' performance. Different parameters needed to be tuned for each model to achieve optimal performance. The parameters in the matrix factorization model were set to reduce the RMSE when providing recommendations to the cold users. In addition, the parameters of the Deep Q-learning models for both approaches, were set to maximize the reward for each taken action. It is also important to note that, all the models were run on an Asus laptop, with an Intel Core i7-6700HQ CPU at 2.60GHz, an NVIDIA GEFORCE 1060 GPU with 6GB of VRAM, and 8GB of RAM and using the Python programming language.

4.3.1. Matrix Factorization Model Parameters

The parameters of the matrix factorization model, namely the number of latent features, the learning rate of the stochastic gradient descent with the Adam optimizer, the number of iterations, and the regularization parameter, were tuned using the 10-fold cross-validation method that was previously mentioned with a sample of 200,000 random observations to speed up this process. More specifically, the number of the latent features was set to 10, meaning that 10 latent features were able to explain the user-item interaction matrix. Moreover, the learning rate that will be used with the stochastic gradient descent optimization method was set to 0.001 and the regularization parameter was set to 0.01. Finally, the number of iterations was set to 100. These hyperparameters can be viewed in Table 1.

Table 1: Matrix Factorization Hyperparameters.

Hyperparameter	**Range**	**Value**
Latent features	[10, 50, 100, 150, 200, 300]	10
Learning rate α	[0.001, 0.003, 0.005]	0.001
Regularization Parameter	[0.01, 0.02, 0.05]	0.01
Iterations	[50, 100, 150]	100

4.3.2. Combined-Heuristic Weights

For the creation of the combined-heuristic AL strategies, namely the PopEnt, PopGini, PopError, and the PopVar strategies, the weights of each of the strategies need to be configured. These weights were configured using a range of possible values ranging from 0 to 1 with a step of 0.1 for each of the strategies, and the values that resulted in the lowest possible RMSE were chosen. Similarly to the research conducted by [16], scenarios with 10 and 100 shown items were considered, and the mean RMSE was taken to conclude to a final value for the weights. The final values that were assigned for the combined-heuristic AL strategies were 0.9 for the Popularity strategy component and 1 for the second strategy (i.e., Entropy, Gini, Error, and Variance).

4.3.3. Deep Q-learning Model Parameters

The hyperparameters of the Deep Q-learning model were set, based on the performance of various hyperparameter combinations. Several different values were tried and evaluated for the learning rate, the γ value, and the ϵ value to determine the optimal values that provided a stable model with a small RMSE. This RMSE value is the mean RMSE value over the total number of episodes. At the beginning of the training procedure, the agent only explored, selecting random actions and having a reward as feedback. This means that initially, the ϵ value is 1. After 50 steps, where the agent only explores the environment, there is a decay value on the exploration rate that decreases the ϵ value, therefore allowing the agent to be trained on previous observations. This is derived by the batch size, meaning the number of samples that will be taken from the transition matrix in order for the agent to be trained. This value was set to 32, meaning that when the agent used previous experience instead of exploration, it used 32 transitions using the Prioritized Experience Replay technique explained in Section 3.3.2. The buffer size, which is the total number of saved transitions, was set to 100, meaning that from those 100 saved transitions, only 32 will be sampled for the training procedure. Next, the discount factor γ was set to 0.99 and the learning rate α to 0.0004. The decay value of the ϵ (ϵ-decay) was set, based on the final exploration rate of the agent. As the final exploration rate ϵ was set to 0.01, meaning that after a specific number of episodes, which was set to 2,000, the ϵ will not decay. An important thing to note here, is that in this case, an episode is defined when a state includes the complete number of shown items for each scenario. Based on these values, the decay of the ϵ was determined to be 0.006 for the agent to be allowed to explore different actions, and finally, the number of steps C after which the target network would be updated was set to 100. Table 2 summarizes the hyperparameters for the DQN model.

Table 2: DQN Hyperparameters.

Hyperparameter	**Range**	**Value**
γ	[0.95, 0.99]	0.99
α	[0.001, 0.0001, 0.0004]	0.0004
ϵ-decay	-	0.00046
C	[50, 100]	100
Buffer Size	[50, 100]	100
Batch Size	[32, 64]	32
Steps until training	-	50
Final ϵ	-	0.01

5. Results

In this section, the results of our research will be compared to the results of the research made by [16] and how the AL strategies they utilized perform in comparison to the different RL approaches as well as the additional AL strategies, namely the Error, PopError, Variance, and PopVar.

Table 3 shows the results for each number of shown items for the RL approaches based on 2,000 episodes, the additional AL strategies, as well as the results of the AL ranking strategies taken from [16]. The best RMSE for each shown items scenario is shown in bold.

When 10 items are shown to the user, it appears that the Random strategy still prevails in terms of RMSE compared to the other AL strategies and the RL approaches. The Random strategy achieves an RMSE of 0.378 which is much lower than that of the other strategies. However, as this strategy proposes random items to the users, it is difficult to say if it will still manage to achieve the same result if it is replicated several times. The Variance strategy manages to achieve the second best result with an RMSE of 0.426 in that case, surpassing the PopGini strategy and followed by the item-based RL approach.

Next, for the 25 and the 50 shown items scenario, the item-based RL approach seems to be the best-performing strategy out of all the AL strategies and the AL-based RL approach. More specifically, in the 25 shown items scenario, the item-based RL approach, having an RMSE of 0.428, surpasses the PopGini

Table 3: Results of the RL approaches and AL strategies.

Strategies	**Shown Items**				
	10	25	50	100	Mean
1. Item-based RL Approach	0.431	**0.428**	**0.421**	0.442	**0.430**
2. AL-based RL Approach	0.467	0.459	0.454	0.453	0.458
3. Random strategy	**0.378**	0.453	0.448	0.466	0.436
4. Popularity strategy	0.448	0.438	0.430	0.422	0.435
5. Gini strategy	0.499	0.490	0.541	0.500	0.501
6. Entropy strategy	0.500	0.488	0.510	0.501	0.500
7. Error strategy	0.536	0.513	0.507	0.475	0.508
8. Variance strategy	0.426	0.449	0.466	0.482	0.456
9. PopGini strategy	0.445	0.433	0.427	**0.417**	0.431
10. PopEnt strategy	0.447	0.442	0.428	0.422	0.435
11. PopError strategy	0.444	0.437	0.428	0.419	0.432
12. PopVar strategy	0.444	0.438	0.429	0.420	0.433

Table 4: Comparison of the results in Table 3. One-tailed p-values for the two-sample Student-t test on the mean RMSE values of two strategies ($H_0 : \mu_{row} = \mu_{col}$, $H_1 : \mu_{row} < \mu_{col}$).

Strategies	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
1. Item-based RL Approach	-	0.001***	0.393	0.296	0.000***	0.000***	0.001***	0.048**	0.500	0.291	0.418	0.377
2. AL-based RL Approach	0.999	-	0.843	0.995	0.003***	0.000***	0.004***	0.577	0.997	0.994	0.997	0.997
3. Random strategy	0.607	0.157	-	0.533	0.010**	0.010**	0.011**	0.216	0.605	0.528	0.579	0.565
4. Popularity strategy	0.704	0.005***	0.467	-	0.001***	0.000***	0.001***	0.079*	0.681	0.488	0.621	0.587
5. Gini strategy	1.000	0.997	0.990	0.999	-	0.725	0.494	0.990	1.000	0.999	1.000	1.000
6. Entropy strategy	1.000	1.000	0.990	1.000	0.275	-	0.286	0.993	1.000	1.000	1.000	1.000
7. Error strategy	0.999	0.996	0.989	0.999	0.506	0.714	-	0.988	0.999	0.999	0.999	0.999
8. Variance strategy	0.952	0.423	0.784	0.921	0.010**	0.007***	0.012**	-	0.946	0.917	0.939	0.935
9. PopGini strategy	0.500	0.003***	0.395	0.319	0.000***	0.000***	0.001***	0.054*	-	0.313	0.429	0.392
10. PopEnt strategy	0.709	0.006***	0.472	0.512	0.001***	0.000***	0.001***	0.083*	0.687	-	0.629	0.596
11. PopError strategy	0.582	0.003***	0.421	0.379	0.000***	0.000***	0.001***	0.061*	0.571	0.371	-	0.462
12. PopVar strategy	0.623	0.003***	0.435	0.413	0.000***	0.000***	0.001***	0.065*	0.608	0.404	0.538	-

$*$, $**$, $***$: Significant under $\alpha = 0.1$, 0.05, 0.01.

strategy by a difference of 0.005 whereas in the 50 items scenario, the item-based RL approach surpasses the PopGini strategy by a difference of 0.006. Still, in the scenario with 100 shown items, the PopGini strategy appears to have the lowest RMSE with a value of 0.417, although followed closely by the PopError strategy with a difference of 0.002.

Overall, the best-performing strategy according to the RMSE metric, seems to be the item-based RL approach, which includes the 200 most popular items. This approach achieves an RMSE of 0.430 and surpasses the PopGini strategy by a small difference of 0.001.

Furthermore, the AL-based RL approach, which was based on the AL strategies, seemed to perform surprisingly poorly compared to all the other strategies, by having large RMSE values. However, the newly created AL strategies (i.e., Error, PopError, Variance, and PopVar) performed well, and most of the times had results which were close to those of the original AL strategies. From the newly added AL strategies, the combined-heuristic strategies, namely the PopVar and the PopError strategies, seemed to perform better than the single-heuristic strategies, except for the scenario with 10 shown items where the Variance strategy performed better, and closely followed the best-performing Random strategy. A possible explanation of the poor performance of the AL-based RL approach, could be that the agent starts from the top ranking items when choosing items from the set of the AL techniques, causing it to not be able to select from all items in the item pool of each strategy. The agent starts from the top ranking items, as these items give the best results according to each AL strategy. For instance, the items with the highest Entropy have the most diverse ratings according to this strategy. Although each AL strategy computes each strategy's score based on all the items (e.g., the Entropy strategy computes the Entropy score for all the items), if the number of shown items is 10, then the maximum number of items that can be shown from each strategy is 10.

To formally compare the results in Table 3, we perform two-sample Student-t tests between the mean RMSE values of all strategies. The results of these tests can be found in Table 4, which provides p-values for the one-tailed test with $H_0 : \mu_{row} = \mu_{col}$ and $H_1 : \mu_{row} < \mu_{col}$, i.e., the alternative hypothesis is accepted in favor of the null hypothesis if the 'row strategy' performs *better* than the 'column strategy' (RMSE is lower). Under a significance level of 0.05, we see that the item-based RL approach outperforms more strategies than any other strategy.

Moreover, as can be seen from Table 5, if the minimum RMSE values that are reached by the RL approaches are taken into consideration, then the item-based RL approach surpasses by a large difference all the other AL strategies in all scenarios.

In most of the scenarios, the RMSE of the item-based RL approach is below 0.4, surpassing also the AL-based RL approach that is based on the AL strategies. Similarly, in this situation, the AL-based RL approach fails to show better results compared to the individual AL ranking strategies. Moreover, the item-based RL approach outperforms the AL-based RL approach by almost 22% in an average scenario, and by up to 35% in the scenario of 10 items shown.

Table 5: Minimum RMSE reached by the RL approaches.

Strategies	**Shown Items**				
	10	25	50	100	Mean
1. Item-based RL Approach	**0.335**	**0.355**	**0.370**	**0.404**	**0.366**
2. AL-based RL Approach	0.451	0.445	0.443	0.443	0.446

A strange phenomenon that appears here, is that in the item-based RL approach, the RMSE value seems to increase as the number of shown items increases. However, this can be explained as the values that are taken here are the minimum values reached for a total of 2,000 episodes for each scenario. Having more episodes could probably rectify this phenomenon. For the training of the agents, the computational time for each model to run was approximately 30 minutes using 2,000 episodes, after which the agent would be terminated and the final results would be shown. To conclude, this means that the items that provide the minimum RMSE value can be selected and suggested to the users to elicit their preferences.

5.1. Software

The creation of all the models, both the matrix factorization as well as the RL models and their visualizations were created solely in Python 3. The Python programming language was chosen as it is the most used language for deep learning and RL applications, and several frameworks exist both for recommender systems as well as for deep/reinforcement learning. For the creation of the matrix factorization model and tuning of the hyperparameters, the Surprise Python framework was used in combination with the Scikit-Learn framework to perform the k-fold cross-validation. After the matrix factorization model was created and the hyperparameters were tuned, the RL models had to be created. For those models, two RL environments had to be built for the agents to operate in. These environments were linked to the agents using the OpenAI Gym framework, and the DQN agents were created using the Stable-Baselines framework, although several other frameworks such as Dopamine and the Keras-RL were available. However, due to the ease of use, the Stable-Baselines was chosen.

Evaluation of the RL models was done internally with the Stable-Baselines framework. Last, the creation of the additional AL strategies was done in Python 2, using the GraphLab Create framework which includes data structures that allow faster computation of the recommender models. The reason for choosing version 2 of Python for the additional AL strategies, was because GraphLab Create is only compatible with Python 2. To sum up, the scores and the results of the additional AL strategies were computed in Python 2 using the GraphLab Create framework whereas the RL agent and the OpenAI Gym environments were created solely in Python 3 using the previously mentioned frameworks. All experiments were performed on a Windows 10 machine with an Intel Core i7 6700HQ 2.60GHz, GTX1060, 8GB RAM, 256GB SSD, and 1TB HDD. The approximate training time of the models was 4 hours. However, because the theoretical complexity of the DQN model is cubic in the number of states, training times differ between the two approaches considered (as the number of states is different).

6. Conclusion

In this work, we proposed a novel approach for addressing the cold-user problem in recommender systems by utilizing state-of-the-art techniques from the fields of RL and AL. The goal of this research was to be able to effectively provide personalized recommendations to the users of a webshop without the use of additional user information. For this task, we applied a Deep Q-learning Network to find the best items to be shown to the users, and then, using those items, provide personalized recommendations.

We evaluated our approach using an implicit feedback dataset from a large Dutch webshop, and compared it to different techniques used by [16]. From the results, it is clear that the item-based RL approach provides the best results compared to several AL strategies as well as compared to the AL-based RL approach. The item-based RL approach outperforms its AL-based counterpart by almost 22% on average, and by almost 35% in the smallest case of 10 shown items. Indeed, the item-based approach was one of the few approaches that seemed to perform better in a scenario with fewer items shown, although a reasonable explanation could be that with a greater number than 2,000 episodes, this phenomenon could have been rectified.

Some avenues for further research may be considered. First, we would like to explore the effects on performance of loss functions other than the Huber loss function considered for DQN. For example, the pseudo Huber loss function provides a smooth approximation to the Huber loss function [51], where the smoothness can be controlled by the researcher. In [51], the improved performance of several generalizations of existing loss functions, among which the pseudo Huber loss function, was shown based on computer vision tasks. It would be interesting to research the robustness of these loss functions in training DQN.

Second, in future research we wish to consider more non-personalized AL strategies for our second RL approach. In particular, we would like to consider the single-heuristic entropy0 (uncertainty based), greedy extend (error reduction), and co-average (attention based), and static-combined heuristic rand-popularity strategies [20] for our second RL approach to test which strategy may improve the performance of our AL-based solution even more.

Third, with the creation of the customized RL environment for the cold-user problem, several other RL techniques can be used besides DQN. More specifically, advancements have been made in the field of RL, and techniques like *Rainbow* [52] have emerged that show promising results. In the future, we plan to experiment with these emerging techniques for addressing the cold-user problem in recommender systems.

Fourth and last, we would like to compare our approaches against deep learning models for addressing the cold-start problem in recommender systems. Among those, generative models stand out, which have shown good performance for the considered problem [53, 54, 55, 56].

References

[1] M. C. Willemsen, M. P. Graus, and B. P. Knijnenburg, "Understanding the role of latent feature diversification on choice difficulty and satisfaction," *User Modeling and User-Adapted Interaction*, vol. 26, no. 4, pp. 347–389, 2016.

[2] S. Iyengar and M. Lepper, "When choice is demotivating: Can one desire too much of a good thing?," *Journal of Personality and Social Psychology*, vol. 79, pp. 995–1006, 2001.

[3] B. Schwartz, "The tyranny of choice," *Scientific American*, vol. 290, pp. 70–5, 2004.

[4] M. J. Pazzani, "A framework for collaborative, content-based and demographic filtering," *Artificial Intelligence Review*, vol. 13, no. 5, pp. 393–408, 1999.

[5] J. Golbeck and J. Hendler, "Filmtrust: movie recommendations using trust in web-based social networks," in *Proceedings of the 3rd IEEE Consumer Communications and Networking Conference (CCNC 2006)*, vol. 1, pp. 282–286, IEEE, 2006.

[6] I. Palomares, F. Browne, and P. Davis, "Multi-view fuzzy information fusion in collaborative filtering recommender systems: Application to the urban resilience domain," *Data & Knowledge Engineering*, vol. 113, pp. 64–80, 2018.

[7] D. K. Panda and S. Ray, "Approaches and algorithms to mitigate cold start problems in recommender systems: a systematic literature review," *Journal of Intelligent Information Systems*, vol. 59, no. 2, pp. 341–366, 2022.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing Atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[9] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[10] K. Li, T. Zhang, R. Wang, Y. Wang, Y. Han, and L. Wang, "Deep reinforcement learning for combinatorial optimization: Covering salesman problems," *IEEE Transactions on Cybernetics*, pp. 1–14, 2021.

[11] X. Wang, Y. Wang, D. Hsu, and Y. Wang, "Exploration in interactive personalized music recommendation: A reinforcement learning approach," *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 11, no. 1, pp. 7:1–7:22, 2014.

[12] X. Zhao, L. Xia, L. Zhang, Z. Ding, D. Yin, and J. Tang, "Deep reinforcement learning for page-wise recommendations," in *Proceedings of the 12th ACM Conference on Recommender Systems (RecSys 2018)*, pp. 95–103, ACM, 2018.

[13] H. V. Dureddy and Z. Kaden, "Handling cold-start collaborative filtering with reinforcement learning," *CoRR*, vol. abs/1806.06192, 2018.

[14] M. Elahi, F. Ricci, and N. Rubens, "A survey of active learning in collaborative filtering recommender systems," *Computer Science Review*, vol. 20, pp. 29–50, 2016.

[15] T. Geurts, S. Giannikis, and F. Frasincar, "Active learning strategies for solving the cold user problem in model-based recommender systems," *Web Intelligence*, vol. 18, no. 4, pp. 269–283, 2020.

[16] T. Geurts and F. Frasincar, "Addressing the cold user problem for model-based recommender systems," in *Proceedings of the 2017 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2017)*, pp. 745–752, ACM, 2017.

[17] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Computing Surveys*, vol. 52, no. 1, pp. 5:1–5:38, 2019.

[18] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez, "Recommender systems survey," *Knowledge-Based Systems*, vol. 46, pp. 109–132, 2013.

[19] H. Wang, N. Wang, and D. Yeung, "Collaborative deep learning for recommender systems," in *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2015)*, pp. 1235–1244, ACM, 2015.

[20] M. Elahi, F. Ricci, and N. Rubens, "A survey of active learning in collaborative filtering recommender systems," *Computer Science Review*, vol. 20, pp. 29–50, 2016.

[21] B. Smith and G. Linden, "Two decades of recommender systems at amazon.com," *IEEE Internet Computing*, vol. 21, no. 03, pp. 12–18, 2017.

[22] G. Linden, J. Jacobi, and E. Benson, "Collaborative recommendations using item-to-item similarity mappings," 1998. patent no. US 6.266.649.

[23] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008)*, pp. 263–272, 2008.

[24] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *IEEE Computer Society*, no. 8, pp. 30–37, 2009.

[25] J. Feng, Z. Xia, X. Feng, and J. Peng, "RBPR: A hybrid model for the new user cold start problem in recommender systems," *Knowledge-Based Systems*, vol. 214, p. 106732, 2021.

[26] R. D. Burke, "Hybrid web recommender systems," in *The Adaptive Web* (P. Brusilovsky, A. Kobsa, and W. Nejdl, eds.), vol. 4321 of *Lecture Notes in Computer Science*, pp. 377–408, Springer, 2007.

[27] R. Salakhutdinov, A. Mnih, and G. Hinton, "Restricted Boltzmann machines for collaborative filtering," in *Proceedings of the 24th International Conference on Machine Learning (ICML 2007)*, pp. 791–798, ACM, 2007.

[28] S. Funk, "Netflix update: Try this at home," *https://sifter.org/simon/journal/20061211.html*, 2006.

[29] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[30] A. Paterek, "Improving regularized singular value decomposition for collaborative filtering," in *Proceedings of KDD Cup and Workshop*, pp. 39–42, ACM, 2007.

[31] R. Salakhutdinov and A. Mnih, "Bayesian probabilistic matrix factorization using markov chain monte carlo," in *Proceedings of the 25th International Conference on Machine Learning (ICML 2008)*, pp. 880–887, ACM, 2008.

[32] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Proceedings of the 13th International Conference on Neural Information Processing Systems (NIPS 2000)*, pp. 535–541, MIT Press, 2000.

[33] F. Ricci, L. Rokach, and B. Shapira, *Recommender Systems Handbook.* Springer, 2nd ed., 2015.

[34] E. Riebe, M. Wright, P. Stern, and B. Sharp, "How to grow a brand: Retain or acquire customers?," *Journal of Business Research*, vol. 67, no. 5, pp. 990–997, 2014.

[35] K. Zhou, S.-H. Yang, and H. Zha, "Functional matrix factorizations for cold-start recommendation," in *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2011)*, pp. 315–324, ACM, 2011.

[36] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *CoRR*, vol. abs/1712.01815, 2017.

[37] X. Tang, Y. Chen, X. Li, J. Liu, and Z. Ying, "A reinforcement learning approach to personalized learning recommendation systems," *British Journal of Mathematical and Statistical Psychology*, vol. 72, no. 1, pp. 108–135, 2019.

[38] Y. Lin, F. Lin, W. Zeng, J. Xiahou, L. Li, P. Wu, Y. Liu, and C. Miao, "Hierarchical reinforcement learning with dynamic recurrent mechanism for course recommendation," *Knowledge-Based Systems*, vol. 244, p. 108546, 2022.

[39] L. Huang, M. Fu, F. Li, H. Qu, Y. Liu, and W. Chen, "A deep reinforcement learning based long-term recommender system," *Knowledge-Based Systems*, vol. 213, p. 106706, 2021.

[40] R. S. Sutton and A. G. Barto, *Reinforcement Learning An Introduction.* MIT Press, 2nd ed., 2018.

[41] S. Choi, H. Ha, U. Hwang, C. Kim, J. Ha, and S. Yoon, "Reinforcement learning based recommender system using biclustering technique," *CoRR*, vol. abs/1801.05532, 2018.

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

[43] G. A. Rummery and M. Niranjan, "On-line Q-learning using connectionist systems," tech. rep., Cambridge University, 1994.

[44] N. Golbandi, Y. Koren, and R. Lempel, "Adaptive bootstrapping of recommender systems using decision trees," in *Proceedings of the 4th ACM International Conference on Web Search and Data Mining (WSDM 2011)*, pp. 595–604, ACM, 2011.

[45] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *Proceedings of the 4th International Conference on Learning Representations (ICLR 2016)*, OpenReview.net, 2016.

[46] Y. Yu, C. Wang, H. Wang, and Y. Gao, "Attributes coupling based matrix factorization for item recommendation," *Applied Intelligence*, vol. 46, no. 3, pp. 521–533, 2017.

[47] C. A. Q. Rana, H. Salima, F. Usama, and C. Hammam, "From a "cold" to a "warm" start in recommender systems," in *Proceedings of the 23rd IEEE International WETICE Conference (WETICE 2014)*, pp. 290–292, IEEE, 2014.

[48] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)*, 2015.

[49] A. Haider, G. Hawe, H. Wang, and B. Scotney, "Gaussian based non-linear function approximation for reinforcement learning," *SN Computer Science*, vol. 2, no. 3, p. 223, 2021.

[50] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. G Bellemare, A. Graves, M. Riedmiller, A. K Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–33, 2015.

[51] J. T. Barron, "A general and adaptive robust loss function," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition 2019 (CVPR 2019)*, pp. 4331–4339, Computer Vision Foundation/IEEE, 2019.

[52] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI 2018)*, pp. 3215–3222, AAAI Press, 2018.

[53] H. Wu, C. Wong, J. Zhang, D. Yu, J. Long, and M. Ng, "Cold-start next-item recommendation by user-item matching and auto-encoders," *IEEE Transactions on Services Computing*, vol. 16, no. 4, pp. 2477–2489, 2023.

[54] H. Bai, M. Hou, L. Wu, Y. Yang, K. Zhang, R. Hong, and M. Wang, "GoRec: A generative cold-start recommendation framework," in *Proceedings of the 31st ACM International Conference on Multimedia (MM 2023)*, pp. 1004–1012, ACM, 2023.

[55] F. Huang, Z. Wang, X. Huang, Y. Qian, Z. Li, and H. Chen, "Aligning distillation for cold-start item recommendation," in *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2023)*, pp. 1147–1157, ACM, 2023.

[56] Z. Zhou, L. Zhang, and N. Yang, "Contrastive collaborative filtering for cold-start item recommendation," in *Proceedings of the ACM Web Conference 2023 (WWW 2023)*, pp. 928–937, ACM, 2023.