# Deep Reinforcement Learning for List-wise Recommendations

Xiangyu Zhao
Data Science and Engineering Lab
Michigan State University
zhaoxi35@msu.edu

Liang Zhang
Data Science Lab
JD.com
zhangliang16@jd.com

Zhuoye Ding
Data Science Lab
JD.com
dingzhuoye@jd.com

Dawei Yin
Data Science Lab
JD.com
yindawei@acm.org

Yihong Zhao
Data Science Lab
JD.com
ericzhao@jd.com

Jiliang Tang
Data Science and Engineering Lab
Michigan State University
tangjili@msu.edu



## ABSTRACT

Recommender systems play a crucial role in mitigating the problem of information overload by suggesting users' personalized items or services. The vast majority of traditional recommender systems consider the recommendation procedure as a static process and make recommendations following a fixed strategy. In this paper, we propose a novel recommender system with the capability of continuously improving its strategies during the interactions with users. We model the sequential interactions between users and a recommender system as a Markov Decision Process (MDP) and leverage Reinforcement Learning (RL) to automatically learn the optimal strategies via recommending trial-and-error items and receiving reinforcements of these items from users' feedbacks. In particular, we introduce an online user-agent interacting environment simulator, which can pre-train and evaluate model parameters offline before applying the model online. Moreover, we validate the importance of list-wise recommendations during the interactions between users and agent, and develop a novel approach to incorporate them into the proposed framework LIRD for list-wide recommendations. The experimental results based on a real-world e-commerce dataset demonstrate the effectiveness of the proposed framework.

## KEYWORDS

List-Wise Recommender System, Deep Reinforcement Learning, Actor-Crtic, Online Environment Simulator.

**ACM Reference Format:**
Xiangyu Zhao, Liang Zhang, Zhuoye Ding, Dawei Yin, Yihong Zhao, and Jiliang Tang. 2018. Deep Reinforcement Learning for List-wise Recommendations. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Recommender systems are intelligent E-commerce applications. They assist users in their information-seeking tasks by suggesting items (products, services, or information) that best fit their needs and preferences. Recommender systems have become increasingly popular in recent years, and have been utilized in a variety of domains including movies, music, books, search queries, and social tags[20, 21]. Most existing recommender systems consider the recommendation procedure as a static process and make recommendations following a fixed greedy strategy. However, these approaches may fail given the dynamic nature of the users' preferences. Furthermore, the majority of existing recommender systems are designed to maximize the immediate (short-term) reward of recommendations, i.e., to make users order the recommended items, while completely overlooking whether these recommended items will lead to more likely or more profitable (long-term) rewards in the future [22].

In this paper, we consider the recommendation procedure as sequential interactions between users and recommender agent; and leverage Reinforcement Learning (RL) to automatically learn the optimal recommendation strategies. Recommender systems based on reinforcement learning have two advantages. First, they are able to continuously update their strategies during the interactions, until the system converges to the optimal strategy that generates recommendations best fitting users' dynamic preferences. Second, the optimal strategy is made by maximizing the expected long-term cumulative reward from users. Therefore, the system can identify the item with a small immediate reward but making big contribution to the rewards for future recommendations.

Efforts have been made on utilizing reinforcement learning for recommender systems, such as POMDP[22] and Q-learning[25]. However, these methods may become inflexible with the increasing number of items for recommendations. This prevents them to be adopted by practical recommender systems. Thus, we leverage Deep Reinforcement Learning[9] with (adapted) artificial neural networks as the non-linear approximators to estimate the action-value function in RL. This model-free reinforcement learning method does not estimate the transition probability and not store the Q-value table. This makes it flexible to support huge amount of items in recommender systems.

### 1.1 List-wise Recommendations

Users in practical recommender systems are typically recommended a list of items at one time. List-wise recommendations are more desired in practice since they allow the systems to provide diverse and complementary options to their users. For list-wise recommendations, we have a list-wise action space, where each action is a set of multiple interdependent sub-actions (items). Existing reinforcement learning recommender methods also could recommend a list of items. For example, DQN[13] can calculate Q-values of all recalled items separately, and recommend a list of items with highest Q-values. However, these approaches recommend items based on one same state, and ignore relationship among the recommended items. As a consequence, the recommended items are similar. In practice, a bundling with complementary items may receive higher rewards than recommending all similar items. For instance, in real-time news feed recommendations, a user may want to read diverse topics of interest, and an action (i.e. recommendation) from the recommender agent would consist of a set of news articles that are not all similar in topics[29]. Therefore, in this paper, we propose a principled approach to capture relationship among recommended items and generate a list of complementary items to enhance the performance.

### 1.2 Architecture Selection

Generally, there exist two Deep Q-learning architectures, shown in Fig.1 (a)(b). Traditional deep Q-learning adopts the first architecture as shown in Fig.1(a), which inputs only the state space and outputs Q-values of all actions. This architecture is suitable for the scenario with high state space and small action space, like playing Atari[13]. However, one drawback is that it cannot handle large and dynamic action space scenario, like recommender systems. The second Q-learning architecture, shown Fig.1(b), treats the state and the action as the input of Neural Networks and outputs the Q-value corresponding to this action. This architecture does not need to store each Q-value in memory and thus can deal with large action space or even continuous action space. A challenging problem of leveraging the second architecture is temporal complexity, i.e., this architecture computes Q-value for all potential actions, separately. To tackle this problem, in this paper, our recommending policy builds upon the Actor-Critic framework[24], shown in Fig.1 (c). The Actor inputs the current state and aims to output the parameters of a state-specific scoring function. Then the RA scores all items and selects an item with the highest score. Next, the Critic uses an approximation architecture to learn a value function (Q-value), which is a judgment of whether the selected action matches the current state. Note that Critic shares the same architecture with the DQN in Fig.1(b). Finally, according to the judgment from Critic, the Actor updates its' policy parameters in a direction of recommending performance improvement to output properer actions in the following iterations. This architecture is suitable for large action space, while can also reduce redundant computation simultaneously.

### 1.3 Online Environment Simulator

Unlike the Deep Q-learning method applied in playing Online Game like Atari, which can take arbitrary action and obtain timely feedback/reward, the online reward is hard to obtain before the recommender system is applied online. In practice, it is necessary to pre-train parameters offline and evaluate the model before applying it online, thus how to train our framework and evaluate the performance of our framework offline is a challenging task. To tackle this challenge, we propose an online environment simulator, which inputs current state and a selected action and outputs a simulated online reward, which enables the framework to train the parameters offline based on the simulated reward. More specifically, we build the simulator by users' historical records. The intuition is no matter what algorithms a recommender system adopt, given the same state ( or a user's historical records) and the same action (recommending the same items to the user), the user will make the same feedbacks to the items.

To evaluate the performance of a recommender system before applying it online, a practical way is to test it based on users' historical clicking/ordering records. However, we only have the ground truth feedbacks (rewards) of the existing items in the users' historical records, which are sparse compared with the enormous item space of current recommender system. Thus we cannot get the feedbacks (rewards) of items that are not in users' historical records. This may result in inconsistent results between offline and online measurements. Our proposed online environment simulator can also mitigate this challenge by producing simulated online rewards given any state-action pair, so that the recommender system can rate items from the whole item space. Based on offline training and evaluation, the well trained parameters can be utilized as the initial parameters when we launch our framework online, which can be updated and improved via on-policy exploitation and exploration.

### 1.4 Our Contributions

We summarize our major contributions as follows:

- We build an online user-agent interacting environment simulator, which is suitable for offline parameters pre-training and evaluation before applying a recommender system online;
- We propose a **LI**st-wise **R**ecommendation framework based on **D**eep reinforcement learning LIRD, which can be applied in scenarios with large and dynamic item space and can reduce redundant computation significantly; and
- We demonstrate the effectiveness of the proposed framework in a real-world e-commerce dataset and validate the importance of list-wise recommendation for accurate recommendations.

The rest of this paper is organized as follows. In Section 2, we first formally define the problem of recommender system via reinforcement learning. Then, we provide approaches to model the recommending procedure as a sequential user-agent interactions and introduce details about employing Actor-Critic framework to automatically learn the optimal recommendation strategies via a online simulator. Section 3 carries out experiments based on real-word e-commerce site and presents experimental results. Section 4

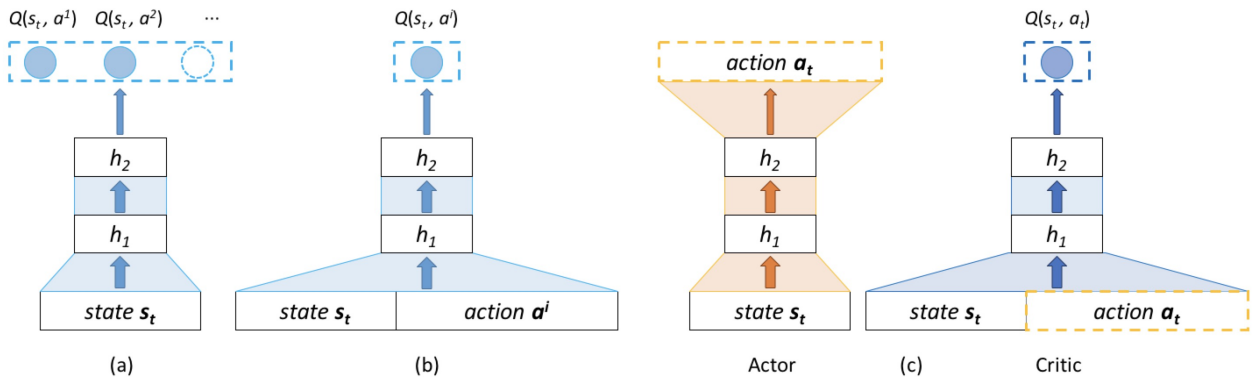


**Figure 1: DQN architecture selection.**

briefly reviews related work. Finally, Section 5 concludes this paper and discusses our future work.

## 2 THE PROPOSED FRAMEWORK

In this section, we first formally define notations and the problem of recommender system via reinforcement learning. Then we build an online user-agent interaction environment simulator. Next, we propose an Actor-Critic based reinforcement learning framework under this setting. Finally, we discuss how to train the framework via users' behavior log and how to utilize the framework for list-wise recommendations.

### 2.1 Problem Statement

We study the recommendation task in which a recommender agent (RA) interacts with environment $\mathcal{E}$ (or users) by sequentially choosing recommendation items over a sequence of time steps, so as to maximize its cumulative reward. We model this problem as a Markov Decision Process (MDP), which includes a sequence of states, actions and rewards. More formally, MDP consists of a tuple of five elements $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ as follows:

- **State space** $\mathcal{S}$: A state $s_t = \{s_t^1, \cdots, s_t^N\} \in \mathcal{S}$ is defined as the browsing history of a user, i.e., previous $N$ items that a user browsed before time $t$. The items in $s_t$ are sorted in chronological order.
- **Action space** $\mathcal{A}$: An action $a_t = \{a_t^1, \cdots, a_t^K\} \in \mathcal{A}$ is to recommend a list of items to a user at time $t$ based on current state $s_t$, where $K$ is the number of items the RA recommends to user each time.
- **Reward** $\mathcal{R}$: After the recommender agent takes an action $a_t$ at the state $s_t$, i.e., recommending a list of items to a user, the user browses these items and provides her feedback. She can skip (not click), click, or order these items, and the agent receives immediate reward $r(s_t, a_t)$ according to the user's feedback.
- **Transition probability** $\mathcal{P}$: Transition probability $p(s_{t+1}|s_t, a_t)$ defines the probability of state transition from $s_t$ to $s_{t+1}$ when RA takes action $a_t$. We assume that the MDP satisfies $p(s_{t+1}|s_t, a_t, ..., s_1, a_1) = p(s_{t+1}|s_t, a_t)$. If user skips all the recommended items, then the next state $s_{t+1} = s_t$; while if the user clicks/orders part of items, then the next state $s_{t+1}$ updates. More details will be shown in following subsections.
- **Discount factor** $\gamma$: $\gamma \in [0, 1]$ defines the discount factor when we measure the present value of future reward. In particular, when $\gamma = 0$, RA only considers the immediate reward. In other words, when $\gamma = 1$, all future rewards can be counted fully into that of the current action.

In practice, only using discrete indexes to denote items is not sufficient since we cannot know the relations between different items only from indexes. One common way is to use extra information to represent items. For instance, we can use the attribute information like brand, price, sale per month, etc. Instead of extra item information, in this paper, we use the user-agent interaction information, i.e., users' browsing history. We treat each item as a word and the clicked items in one recommendation session as a sentence. Then, we can obtain dense and low-dimensional vector representations for items via word embedding[8].

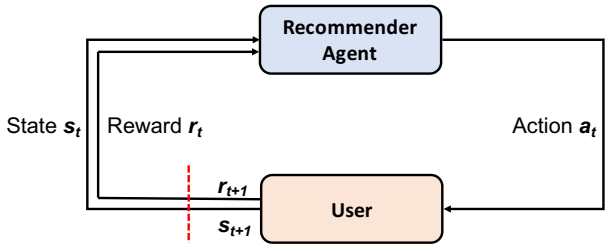


**Figure 2: The agent-user interactions in MDP.**

Figure 2 illustrates the agent-user interactions in MDP. By interacting with the environment (users), recommender agent takes actions (recommends items) to users in such a way that maximizes the expected return, which includes the delayed rewards. We follow the standard assumption that delayed rewards are discounted by a factor of $\gamma$ per time-step.

With the notations and definitions above, the problem of list-wise item recommendation can be formally defined as follows: *Given the historical MDP, i.e., $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, the goal is to find*

*a recommendation policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$, which can maximize the cumulative reward for the recommender system.*

## 2.2 Online User-Agent Interaction Environment Simulator

To tackle the challenge of training our framework and evaluating the performance of our framework offline, in this subsection, we propose an online user-agent interaction environment simulator. In the online recommendation procedure, given the current state $s_t$, the RA recommends a list of items $a_t$ to a user, and the user browses these items and provides her feedbacks, i.e., skip/click/order part of the recommended items. The RA receives immediate reward $r(s_t, a_t)$ according to the user's feedback. To simulate the aforementioned online interaction procedures, the task of simulator is to predict a reward based on current state and a selected action, i.e., $f : (s_t, a_t) \rightarrow r_t$.

According to collaborative filtering techniques, users with similar interests will make similar decisions on the same item. With this intuition, we match the current state and action to existing historical state-action pairs, and stochastically generate a simulated reward. To be more specific, we first build a memory $\mathcal{M} = \{m_1, m_2, \cdots\}$ to store users' historical browsing history, where $m_i$ is a user-agent interaction triple $((s_i, a_i) \rightarrow r_i)$. The procedure to build the online simulator memory is illustrated in Algorithm 1. Given a historical recommendation session $\{a_1, \cdots, a_L\}$, we can observe the initial state $s_0 = \{s_0^1, \cdots, s_0^N\}$ from the previous sessions (line 2). Each time we observe $K$ items in temporal order (line 3), where "$l = 1, L; K$" means that each iteration we will move forward a window of $K$. We can observe the current state (line 4), current $K$ items (line 5), and the user's feedbacks for these items (line 6). Then we store triple $((s, a) \rightarrow r)$ in memory(line-7). Finally we update the state (lines 8-13), and move to the next $K$ items. Since we keep a fixed length state $s = \{s^1, \cdots, s^N\}$, each time a user clicked/ordered some items in the recommended list, we add these items to the end of state and remove the same number of items in the top of the state. For example, the RA recommends a list of five items $\{a_1, \cdots, a_5\}$ to a user, if the user clicks $a_1$ and orders $a_5$, then update $s = \{s^3, \cdots, s^N, a_1, a_5\}$.

Then we calculated the similarity of the current state-action pair, say $p_t(s_t, a_t)$, to each existing historical state-action pair in the memory. In this work, we adopt cosine similarity as:

$$Cosine(p_t, m_i) = \alpha \frac{s_t s_i^\top}{\|s_t\|\|s_i\|} + (1-\alpha)\frac{a_t a_i^\top}{\|a_t\|\|a_i\|}, \tag{1}$$

where the first term measures the state similarity and the second term evaluates the action similarity. Parameter $\alpha$ controls the balance of two similarities. Intuitively, with the increase of similarity between $p_t$ and $m_i$, there is a higher chance $p_t$ mapping to the reward $r_i$. Thus the probability of $p_t \rightarrow r_i$ can be defined as follows:

$$P(p_t \rightarrow r_i) = \frac{Cosine(p_t, m_i)}{\sum_{m_j \in \mathcal{M}} Cosine(p_t, m_j)}, \tag{2}$$

then we can map the current state-action pair $p_t$ to a reward according the above probability. The major challenge of this projection is the computation complexity, i.e., we must compute pair-wise similarity between $p_t$ and each $m_i \in \mathcal{M}$. To tackle this challenge,

**Algorithm 1** Building Online Simulator Memory.

**Input**: Users' historical sessions $B$, and the length of recommendation list $K$.
**Output**:Simulator Memory $\mathcal{M}$

1: **for** *session* $= 1, B$ **do**
2: Observe initial state $s_0 = \{s_0^1, \cdots, s_0^N\}$
3: **for** *item order* $l = 1, L; K$ **do**
4: Observe current state $s = \{s^1, \cdots, s^N\}$
5: Observe current action list $a = \{a_l, \cdots, a_{l+K-1}\}$
6: Observe current reward list $r = \{r_l, \cdots, r_{l+K-1}\}$
7: Add the triple $((s, a) \rightarrow r)$ in $\mathcal{M}$
8: **for** $k = 0, K-1$ **do**
9: **if** $r_{l+k} > 0$ **then**
10: Remove the first item in $s$
11: Add the item $a_{l+k}$ in the bottom of $s$
12: **end if**
13: **end for**
14: **end for**
15: **end for**
16: **return** $\mathcal{M}$

we first group users' historical browsing history according to the rewards. Note that the number of reward permutation is typically limited. For example, the RA recommends two items to user each time, and the reward of user skip/click/order an item is 0/1/5, then the permutation of two items' rewards is 9, i.e., $\mathcal{U} = \{\mathcal{U}_1, \cdots, \mathcal{U}_9\} = \{(0,0), (0,1), (0,5), (1,0), (1,1), (1,5), (5,0), (5,1), (5,5)\}$, which is much smaller than the total number of historical records. Then probability of mapping $p_t$ to $\mathcal{U}_x$ can be computed as follows:

$$\begin{aligned} P(p_t \rightarrow \mathcal{U}_x) &= \frac{\sum_{r_i = \mathcal{U}_x} Cosine(p_t, m_i)}{\sum_{m_j \in \mathcal{M}} Cosine(p_t, m_j)} \\ &= \frac{\alpha \frac{s_t}{\|s_t\|} \cdot \sum\limits_{r_i = \mathcal{U}_x} \frac{s_i^\top}{\|s_i\|} + (1-\alpha)\frac{a_t}{\|a_t\|} \cdot \sum\limits_{r_i = \mathcal{U}_x} \frac{a_i^\top}{\|a_i\|}}{\sum\limits_{\mathcal{U}_y \in \mathcal{U}} \left( \alpha \frac{s_t}{\|s_t\|} \cdot \sum\limits_{r_j = \mathcal{U}_y} \frac{s_j^\top}{\|s_j\|} + (1-\alpha)\frac{a_t}{\|a_t\|} \cdot \sum\limits_{r_j = \mathcal{U}_y} \frac{a_j^\top}{\|a_j\|} \right)} \\ &= \frac{\mathcal{N}_x \cdot \left( \alpha \frac{s_t \bar{s_x}^\top}{\|s_t\|} + (1-\alpha)\frac{a_t \bar{a_x}^\top}{\|a_t\|} \right)}{\sum\limits_{\mathcal{U}_y \in \mathcal{U}} \mathcal{N}_y \cdot \left( \alpha \frac{s_t \bar{s_y}^\top}{\|s_t\|} + (1-\alpha)\frac{a_t \bar{a_y}^\top}{\|a_t\|} \right)} \end{aligned} \tag{3}$$

where we assume that $r_i$ is a reward list containing user's feedbacks of the recommended items, for instance $r_i = (1, 5)$. $\mathcal{N}_x$ is the size of users' historical browsing history group that $r = \mathcal{U}_x$. $\bar{s_x}$ and $\bar{a_x}$ are the average state vector and average action vector for $r = \mathcal{U}_x$, i.e., $\bar{s_x} = \frac{1}{\mathcal{N}_x} \sum_{r_i = \mathcal{U}_x} s_i / \|s_i\|$, and $\bar{a_x} = \frac{1}{\mathcal{N}_x} \sum_{r_i = \mathcal{U}_x} a_i / \|a_i\|$. The simulator only needs to pre-compute the $\mathcal{N}_x$, $\bar{s_x}$ and $\bar{a_x}$, and can map $p_t$ to a reward list $\mathcal{U}_x$ according to the probability in Eq.(3). In practice, RA updates $\mathcal{N}_x$, $\bar{s_x}$ and $\bar{a_x}$ every 1000 episodes. As $|\mathcal{U}|$ is much smaller than the total number of historical records, Eq.(3) can map $p_t$ to a reward list $\mathcal{U}_x$ efficiently.

In practice, the reward is usually a number, rather than a vector. Thus if the $p_t$ is mapped to $\mathcal{U}_x$, we calculate the overall reward $r_t$

of the whole recommended list as follows:

$$r_t = \sum_{k=1}^{K} \Gamma^{k-1} \mathcal{U}_x^k, \tag{4}$$

where $k$ is the order that an item in the recommended list and $K$ is the length of the recommended list, and $\Gamma \in (0, 1]$. The intuition of Eq.(4) is that reward in the top of recommended list has a higher contribution to the overall rewards, which force RA arranging items that user may order in the top of the recommended list.

## 2.3 The Actor Framework

In this subsection, we propose the list-wise item recommending procedure, which consists of two steps, i.e., 1) state-specific scoring function parameter generating, and 2) action generating. Current practical recommender systems rely on a scoring or rating system which is averaged across all users ignoring specific demands of a user. These approaches perform poorly in tasks where there is large variation in users' interests. To tackle this problem, we present a state-specific scoring function, which rates items according to user's current state.

In the previous section, we have defined the state $s$ as the whole browsing history, which can be infinite and inefficient. A better way is to only consider the positive items, e.g., previous 10 clicked/ ordered items. A good recommender system should recommend the items that users prefer the most. The positive items represent key information about users' preferences, i.e., which items the users prefer to. Thus, we only consider them for state-specific scoring function.

Our state-specific scoring function parameter generating step maps the current state $s_t = \{s_t^1, \cdots, s_t^N\}$ to a list of weight vectors $\mathbf{w}_t = \{\mathbf{w}_t^1, \cdots, \mathbf{w}_t^K\}$ as follows:

$$f_{\theta^\pi} : s_t \rightarrow \mathbf{w}_t \tag{5}$$

where $f_{\theta^\pi}$ is a function parametrized by $\theta^\pi$, mapping from the state space to the weight representation space. Here we choose deep neural networks as the parameter generating function.

Next we present the action-generating step based on the aforementioned scoring function parameters. Without the loss of generality, we assume that the scoring function parameter $\mathbf{w}_t^k$ and the embedding $\mathbf{e}_i$ of $i^{th}$ item from the item space $\mathcal{I}$ is linear-related as:

$$socre_i = \mathbf{w}_t^k \mathbf{e}_i^\top. \tag{6}$$

Note that it is straightforward to extend it with non-linear relations. Then after computing scores of all items, the RA selects an item with highest score as the sub-action $a_t^k$ of action $a_t$. We present list-wise item recommendation algorithm in Algorithm 2.

The Actor first generates a list of weight vectors (line 1). For each weight vector, the RA scores all items in the item space (line 3), selects the item with highest score (line 4), and then adds this item at the end of the recommendation list. Finally the RA removes this item from the item space, which prevents recommending the same item to the recommendation list.

## 2.4 The Critic Framework

The Critic is designed to leverage an approximator to learn an action-value function $Q(s_t, a_t)$, which is a judgment of whether the action $a_t$ generated by Actor matches the current state $s_t$. Then, according $Q(s_t, a_t)$, the Actor updates its' parameters in a direction of improving performance to generate proper actions in the following iterations. Many applications in reinforcement learning make use of the optimal action-value function $Q^*(s_t, a_t)$. It is the maximum expected return achievable by the optimal policy, and should follow the Bellman equation [2] as:

**Algorithm 2** List-Wise Item Recommendation Algorithm.

**Input**: Current state $s_t$, Item space $\mathcal{I}$, the length of recommendation list $K$.
**Output**:Recommendation list $a_t$.

1: Generate $\mathbf{w}_t = \{\mathbf{w}_t^1, \cdots, \mathbf{w}_t^K\}$ according Eq.(5)
2: **for** $k = 1, K$ **do**
3: Score items in $\mathcal{I}$ according Eq.(6)
4: Select the an item with highest score as $a_t^k$
5: Add item $a_t^k$ in the bottom of $a_t$
6: Remove item $a_t^k$ from $\mathcal{I}$
7: **end for**
8: **return** $a_t$

$$Q^*(s_t, a_t) = \mathbb{E}_{s_{t+1}}\left[r_t + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) | s_t, a_t\right]. \tag{7}$$

In practice, to select an optimal $a_{t+1}$, $|\mathcal{A}|$ evaluations are necessary for the inner operation max. This prevents Eq.(7) to be adopted in practical recommender systems with the enormous action space. However, the Actor architectures proposed in Section 2.3 outputs a deterministic action for Critic, which avoids the aforementioned computational cost of $|\mathcal{A}|$ evaluations in Eq.(7) as follows:

$$Q(s_t, a_t) = \mathbb{E}_{s_{t+1}}\left[r_t + \gamma Q(s_{t+1}, a_{t+1}) | s_t, a_t\right]. \tag{8}$$

where the Q-value function $Q(s_t, a_t)$ is the expected return based on state $s_t$ and the action $a_t$.

In real recommender systems, the state and action spaces are enormous, thus estimating the action-value function $Q(s, a)$ for each state-action pair is infeasible. In addition, many state-action pairs may not appear in the real trace such that it is hard to update their values. Therefore, it is more flexible and practical to use an approximator function to estimate the action-value function, i.e., $Q(s, a) \approx Q(s, a; \theta^\mu)$ . In practice, the action-value function is usually highly nonlinear. Deep neural networks are known as excellent approximators for non-linear functions. In this paper, We refer to a neural network function approximator with parameters $\theta^\mu$ as deep Q-network (DQN). A DQN can be trained by minimizing a sequence of loss functions $L(\theta^\mu)$ as

$$L(\theta^\mu) = \mathbb{E}_{s_t, a_t, r_t, s_{t+1}}\left[(y_t - Q(s_t, a_t; \theta^\mu))^2\right], \tag{9}$$

where $y_t = \mathbb{E}_{s_{t+1}}[r_t + \gamma Q'(s_{t+1},\ a_{t+1}; \theta^{\mu'})|s_t, a_t]$ is the target for the current iteration. The parameters from the previous iteration $\theta^{\mu'}$ are fixed when optimizing the loss function $L(\theta^\mu)$. In practice, it is often computationally efficient to optimize the loss function by stochastic gradient descent, rather than computing the full expectations in the above gradient.

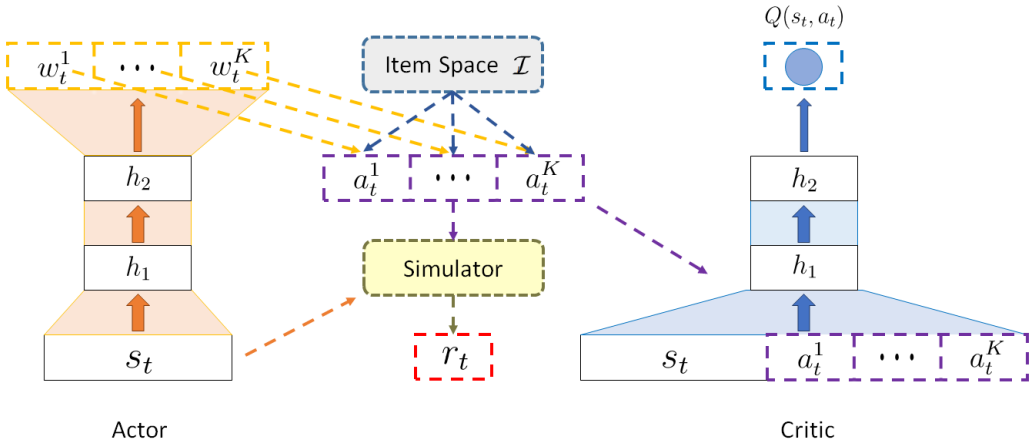


**Figure 3: An illustration of the proposed framework with online simulator.**

## 2.5 The Training Procedure

An illustration of the proposed user-agent online interaction simulator and deep reinforcement recommending LIRD framework is demonstrated in Figure 3. Next, we discuss the parameters training procedures. In this work, we utilize DDPG algorithm[9] to train the parameters of the proposed framework. The training algorithm for the proposed framework DEV is presented in Algorithm 3.

In each iteration, there are two stages, i.e., 1) transition generating stage (lines 8-20), and 2) parameter updating stage (lines 21-28). For transition generating stage (line 8): given the current state $s_t$, the RA first recommends a list of items $a_t = \{a_t^1, \cdots, a_t^K\}$ according to Algorithm 2 (line 9); then the agent observes the reward $r_t$ from simulator (line 10) and updates the state to $s_{t+1}$ (lines 11-17) following the same strategy in Algorithm 1; and finally the recommender agent stores transitions $(s_t, a_t, r_t, s_{t+1})$ into the memory $\mathcal{D}$ (line 19), and set $s_t = s_{t+1}$(line 20). For parameter updating stage: the recommender agent samples mini-batch of transitions $(s, a, r, s')$ from $\mathcal{D}$ (line 22), and then updates parameters of Actor and Critic (lines 23-28) following a standard DDPG procedure [9].

In the algorithm, we introduce widely used techniques to train our framework. For example, we utilize a technique known as *experience replay* [10] (lines 3,22), and introduce separated evaluation and target networks [13](lines 2,23), which can help smooth the learning and avoid the divergence of parameters. For the soft target updates of target networks(lines 27,28), we used $\tau = 0.001$. Moreover, we leverage *prioritized sampling strategy* [15] to assist the framework learning from the most important historical transitions.

## 2.6 The Testing Procedure

After framework training stage, RA gets well-trained parameters, say $\Theta^\pi$ and $\Theta^\mu$. Then we can do framework testing on simulator environment. The model testing also follows Algorithm 3, i.e., the parameters continuously updates during the testing stage, while the major difference from training stage is before each recommendation session, we reset the parameters back to $\Theta^\pi$ and $\Theta^\mu$, for the sake of fair comparison between each session. We can artificially control the length of recommendation session to study the short-term and long-term performance.

# 3 EXPERIMENTS

In this section, we conduct extensive experiments with a dataset from a real e-commerce site to evaluate the effectiveness of the proposed framework. We mainly focus on two questions: (1) how the proposed framework performs compared to representative baselines; and (2) how the list-wise strategy contributes to the performance. We first introduce experimental settings. Then we seek answers to the above two questions. Finally, we study the impact of important parameters on the performance of the proposed framework.

## 3.1 Experimental Settings

We evaluate our method on a dataset of July, 2017 from a real e-commerce site. We randomly collect 100,000 recommendation sessions (1,156,675 items) in temporal order, and use the first 70% sessions as the training set and the later 30% sessions as the testing set. For a given session, the initial state is collected from the previous sessions of the user. In this paper, we leverage $N = 10$ previously clicked/ordered items as the positive state. Each time the RA recommends a list of $K = 4$ items to users. The reward $r$ of skipped/clicked/ordered items are empirically set as 0, 1, and 5, respectively. The dimension of the item embedding is 50, and we set the discounted factor $\gamma = 0.75$. For the parameters of the proposed framework such as $K$ and $\gamma$, we select them via cross-validation. Correspondingly, we also do parameter-tuning for baselines for a fair comparison. We will discuss more details about parameter selection for the proposed framework in the following subsections.

To evaluate the performance of the proposed framework, we select **MAP** [27] and **NDCG** [6] as the metrics to measure the performance. The difference of ours from traditional Learn-to-Rank methods is that we rank both clicked and ordered items together, and set them by different rewards, rather than only rank clicked items as that in Learn-to-Rank problems.

**Algorithm 3** Parameters Training for DEV with DDPG.

Initialize actor network $f_{\theta^\pi}$ and critic network $Q(s, a|\theta^\mu)$ with random weights
Initialize target network $f'$ and $Q'$ with weights $\theta^{\pi'} \leftarrow \theta^\pi, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize the capacity of replay memory $\mathcal{D}$
**for** *session* $= 1, M$ **do**
  Reset the item space $\mathcal{I}$
  Initialize state $s_0$ from previous sessions
  **for** $t = 1, T$ **do**
    **Stage 1: Transition Generating Stage**
    Select an action $a_t = \{a_t^1, \cdots, a_t^K\}$ according **Alg.2**
    Execute action $a_t$ and observe the reward list $\{r_t^1, \cdots, r_t^K\}$ for each item in $a_t$
    Set $s_{t+1} = s_t$
    **for** $k = 1, K$ **do**
      **if** $r_t^k > 0$ **then**
        Add $a_t^k$ to the end of $s_{t+1}$
        Remove the first item of $s_{t+1}$
      **end if**
    **end for**
    Compute overall reward $r_t$ according Eq. (4)
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$
    Set $s_t = s_{t+1}$
    **Stage 2: Parameter Updating Stage**
    Sample minibatch of $\mathcal{N}$ transitions $(s, a, r, s')$ from $\mathcal{D}$
    Generate $a'$ by target Actor network according **Alg.2**
    Set $y = r + \gamma Q'(s', a'; \theta^{\mu'})$
    Update Critic by minimizing $\big(y - Q(s, a; \theta^\mu)\big)^2$ according to:

$$\nabla_{\theta^\mu} L(\theta^\mu) \approx \frac{1}{\mathcal{N}} \big[(y - Q(s, a; \theta^\mu)) \nabla_{\theta^\mu} Q(s, a; \theta^\mu)\big]$$

    Update the Actor using the sampled policy gradient:

$$\nabla_{\theta^\pi} f_{\theta^\pi} \approx \frac{1}{\mathcal{N}} \sum_i \nabla_a Q(s, a|\theta^\mu) \nabla_{\theta^\pi} f_{\theta^\pi}(s)$$

    Update the Critic target networks:

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1 - \tau)\theta^{\mu'}$$

    Update the Actor target networks:

$$\theta^{\pi'} \leftarrow \tau\theta^\pi + (1 - \tau)\theta^{\pi'}$$

  **end for**
**end for**

## 3.2 Performance Comparison for Item Recommendations

To answer the the first question, we compare the proposed framework with the following representative baseline methods:

- **CF**: Collaborative filtering[3] is a method of making automatic predictions about the interests of a user by collecting preference information from many users, which is based on the hypothesis that people often get the best recommendations from someone with similar tastes to themselves.
- **FM**: Factorization Machines[19] combine the advantages of support vector machines with factorization models. Compared with matrix factorization, higher order interactions can be modeled using the dimensionality parameter.
- **DNN**: We choose a deep neural network with back propagation technique as a baseline to recommend the items in a given session. The input of DNN is the embeddings of users' historical clicked/ordered items. We train the DNN to output the next recommended item.
- **RNN**: This baseline utilizes the basic RNN to predict what user will buy next based on the clicking/ordering histories. To minimize the computation costs, it only keeps a finite number of the latest states.
- **DQN**: We use a Deep Q-network[13] with embeddings of users' historical clicked/ordered items (state) and a recommended item (action) as input, and train this baseline following Eq. 7. Note that the DQN shares the same architecture with the Critic in our framework.

## 3.3 Performance of List-Wise Recommendations

To validate the effectiveness of the list-wise recommendation strategy, we investigate how the proposed framework LIRD performs with the changes of the length of the recommendation list, i.e., $K$, in long-term sessions, while fixing other parameters. Note that $K = 1$ is the item-wise recommendation.

## 3.4 Performance of Simulator

The online simulator has one key parameter, i.e., $\alpha$, which controls the trade-off between state and action similarity in simulator, see Eq.(3). To study the impact of this parameter, we investigate how the proposed framework LIRD works with the changes of $\alpha$ in long-term sessions, while fixing other parameters.

# 4 RELATED WORK

In this section, we briefly review works related to our study. In general, the related work can be mainly grouped into the following categories.

The first category related to this paper is traditional recommendation techniques. Recommender systems assist users by supplying a list of items that might interest users. Efforts have been made on offering meaningful recommendations to users. Collaborative filtering[11] is the most successful and the most widely used technique, which is based on the hypothesis that people often get the best recommendations from someone with similar tastes to themselves[3]. Another common approach is content-based filtering[14], which tries to recommend items with similar properties to those that a user ordered in the past. Knowledge-based systems[1] recommend items based on specific domain knowledge about how certain item features meet usersâĂŹ needs and preferences and how the item is useful for the user. Hybrid recommender systems are based on the combination of the above mentioned two or more types of techniques[4]. The other topic closely related to this category is deep learning based recommender system, which is able to effectively capture the non-linear and non-trivial user-item

relationships, and enables the codification of more complex abstractions as data representations in the higher layers[30]. For instance, Nguyen et al.[17] proposed a personalized tag recommender system based on CNN. It utilizes constitutional and max-pooling layer to get visual features from patches of images. Wu et al.[28] designed a session-based recommendation model for real-world e-commerce website. It utilizes the basic RNN to predict what user will buy next based on the click histories. This method helps balance the tradeoff between computation costs and prediction accuracy.

The second category is about reinforcement learning for recommendations, which is different with the traditional item recommendations. In this paper, we consider the recommending procedure as sequential interactions between users and recommender agent; and leverage reinforcement learning to automatically learn the optimal recommendation strategies. Indeed, reinforcement learning have been widely examined in recommendation field. The MDP-Based CF model in Shani et al.[22] can be viewed as approximating a partial observable MDP (POMDP) by using a finite rather than unbounded window of past history to define the current state. To reduce the high computational and representational complexity of POMDP, three strategies have been developed: value function approximation[5], policy based optimization [16, 18], and stochastic sampling [7]. Furthermore, Mahmood et al.[12] adopted the reinforcement learning technique to observe the responses of users in a conversational recommender, with the aim to maximize a numerical cumulative reward function modeling the benefit that users get from each recommendation session. Taghipour et al.[25, 26] modeled web page recommendation as a Q-Learning problem and learned to make recommendations from web usage data as the actions rather than discovering explicit patterns from the data. The system inherits the intrinsic characteristic of reinforcement learning which is in a constant learning process. Sunehag et al.[23] introduced agents that successfully address sequential decision problems with high-dimensional combinatorial slate-action spaces.

## 5 CONCLUSION

In this paper, we propose a novel framework LIRD, which models the recommendation session as a Markov Decision Process and leverages Deep Reinforcement Learning to automatically learn the optimal recommendation strategies. Reinforcement learning based recommender systems have two advantages: (1) they can continuously update strategies during the interactions, and (2) they are able to learn a strategy that maximizes the long-term cumulative reward from users. Different from previous work, we propose a list- wise recommendation framework, which can be applied in scenarios with large and dynamic item space and can reduce redundant computation significantly. Note that we design an online user-agent interacting environment simulator, which is suitable for offline parameters pre-training and evaluation before applying a recommender system online. We evaluate our framework with extensive experiments based on data from a real e-commerce site. The results show that (1) our framework can improve the recommendation performance; and (2) list-wise strategy outperforms item-wise strategies.

There are several interesting research directions. First, in addition to positional order of items we used in this work, we would like to investigate more orders like temporal order. Second, we would like to validate with more agent-user interaction patterns, e.g., adding items into shopping cart, and investigate how to model them mathematically for recommendations. Finally, the framework proposed in the work is quite general, and we would like to investigate more applications of the proposed framework, especially for those applications with both positive and negative(skip) signals.

## REFERENCES


[1] Rajendra Akerkar and Priti Sajja. 2010. *Knowledge-based systems*. Jones & Bartlett Publishers.

[2] Richard Bellman. 2013. *Dynamic programming*. Courier Corporation.

[3] John S Breese, David Heckerman, and Carl Kadie. 1998. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 43–52.

[4] Robin Burke. 2002. Hybrid recommender systems: Survey and experiments. *User modeling and user-adapted interaction* 12, 4 (2002), 331–370.

[5] Milos Hauskrecht. 1997. Incremental methods for computing bounds in partially observable Markov decision processes. In *AAAI/IAAI*. 734–739.

[6] Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)* 20, 4 (2002), 422–446.

[7] Michael Kearns, Yishay Mansour, and Andrew Y Ng. 2002. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine learning* 49, 2 (2002), 193–208.

[8] Omer Levy and Yoav Goldberg. 2014. Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*. 2177–2185.

[9] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[10] Long-Ji Lin. 1993. *Reinforcement learning for robots using neural networks*. Technical Report. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

[11] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing* 7, 1 (2003), 76–80.

[12] Tariq Mahmood and Francesco Ricci. 2009. Improving recommender systems with adaptive conversational strategies. In *Proceedings of the 20th ACM conference on Hypertext and hypermedia*. ACM, 73–82.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[14] Raymond J Mooney and Loriene Roy. 2000. Content-based book recommending using learning for text categorization. In *Proceedings of the fifth ACM conference on Digital libraries*. ACM, 195–204.

[15] Andrew W Moore and Christopher G Atkeson. 1993. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine learning* 13, 1 (1993), 103–130.

[16] Andrew Y Ng and Michael Jordan. 2000. PEGASUS: A policy search method for large MDPs and POMDPs. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 406–415.

[17] Hanh TH Nguyen, Martin Wistuba, Josif Grabocka, Lucas Rego Drumond, and Lars Schmidt-Thieme. 2017. Personalized Deep Learning for Tag Recommendation. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 186–197.

[18] Pascal Poupart and Craig Boutilier. 2005. VDCBPI: an approximate scalable algorithm for large POMDPs. In *Advances in Neural Information Processing Systems*. 1081–1088.

[19] Steffen Rendle. 2010. Factorization machines. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*. IEEE, 995–1000.

[20] Paul Resnick and Hal R Varian. 1997. Recommender systems. *Commun. ACM* 40, 3 (1997), 56–58.

[21] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2011. Introduction to recommender systems handbook. In *Recommender systems handbook*. Springer, 1–35.

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

[23] Peter Sunehag, Richard Evans, Gabriel Dulac-Arnold, Yori Zwols, Daniel Visentin, and Ben Coppin. 2015. Deep Reinforcement Learning with Attention for Slate Markov Decision Processes with High-Dimensional States and Actions. *arXiv preprint arXiv:1512.01124* (2015).

[24] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.

[25] Nima Taghipour and Ahmad Kardan. 2008. A hybrid web recommender system based on q-learning. In *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 1164–1168.

[26] Nima Taghipour, Ahmad Kardan, and Saeed Shiry Ghidary. 2007. Usage-based web recommendations: a reinforcement learning approach. In *Proceedings of the 2007 ACM conference on Recommender systems*. ACM, 113–120.

[27] Andrew Turpin and Falk Scholer. 2006. User performance versus precision measures for simple search tasks. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 11–18.

[28] Sai Wu, Weichao Ren, Chengchao Yu, Gang Chen, Dongxiang Zhang, and Jingbo Zhu. 2016. Personal recommendation using deep recurrent neural networks in NetEase. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE, 1218–1229.

[29] Yisong Yue and Carlos Guestrin. 2011. Linear submodular bandits and their application to diversified retrieval. In *Advances in Neural Information Processing Systems*. 2483–2491.

[30] Shuai Zhang, Lina Yao, and Aixin Sun. 2017. Deep Learning based Recommender System: A Survey and New Perspectives. *arXiv preprint arXiv:1707.07435* (2017).