Pseudo Dyna-Q: A Reinforcement Learning Framework for Interactive Recommendation

Lixin Zou\(^1\), Long Xia\(^2\), Pan Du\(^3\), Zhuo Zhang\(^4\),
Ting Bai\(^5\), Weidong Liu\(^1\), Jian-Yun Nie\(^3\), Dawei Yin\(^6\),\(^*\)
\(^{1}\)Tsinghua University, China, \(^{2}\)York University, Canada
\(^{3}\)University of Montreal, Canada, \(^{4}\)The University of Melbourne, Australia
\(^{5}\)Beijing University of Posts and Telecommunications, China, \(^{6}\)JD Data Science Lab, China
\(^{*}\)Corresponding author.

baiting@bupt.edu.cn, \{zoulx15,liuwd\}@mails.tsinghua.edu.cn, longxia@yorku.ca, zhuo.zhang.cn@gmail.com

ABSTRACT

Applying reinforcement learning (RL) in recommender systems is attractive but costly due to the constraint of the interaction with real customers, where performing online policy learning through interacting with real customers usually harms customer experiences. A practical alternative is to build a recommender agent offline from logged data, whereas directly using logged data offline leads to the problem of selection bias between logging policy and the recommendation policy. The existing direct offline learning algorithms, such as Monte Carlo methods and temporal difference methods are either computationally expensive or unstable on convergence.

To address these issues, we propose **Pseudo Dyna-Q (PDQ)**. In PDQ, instead of interacting with real customers, we resort to a customer simulator, referred to as the World Model, which is designed to simulate the environment and handle the selection bias of logged data. During policy improvement, the World Model is constantly updated and optimized adaptively, according to the current recommendation policy. This way, the proposed PDQ not only avoids the instability of convergence and high computation cost of existing approaches but also provides unlimited interactions without involving real customers. Moreover, a proved upper bound of empirical error of reward function guarantees that the learned offline policy has lower bias and variance. Extensive experiments demonstrated the advantages of PDQ on two real-world datasets against state-of-the-arts methods.

CCS CONCEPTS

- Information systems → Recommender systems; Personalization;
- Theory of computation → Sequential decision making;

KEYWORDS

Pseudo Dyna-Q, Customer Simulator, Model-Based Reinforcement Learning, Offline Policy Learning, Recommender Systems

1 INTRODUCTION

Recommender systems have shown its effectiveness and become more popular for past decades. Recommendation by nature is an interactive process: a recommendation agent suggests items, based on customers’ preferences; customers provide feedback on the suggested items; and the agent updates customers’ preferences and makes further recommendations. Applying reinforcement learning (RL) to interactive recommendation (e.g., personalized music streams in Spotify\(^1\), product feeds in Amazon\(^2\), image feeds in Pinterest\(^3\)) has attracted a lot of interests from the research community [30, 47, 49]. However, employing RL in real-world recommender systems still remains challenging. In general, the RL agents are learned through trial-and-error search, such as Atari games [22] and AlphaGo [31, 32], where the agents improve policy via numerous failures before achieving greatest strides. In realistic recommender systems, directly building a recommender agent from scratch, which requires the agent to interact with real customers numerous times, will hurt customer experiences and no customer would be willing to collaborate for a long time. An alternative is to make use of the logged data and build a recommender agent in offline manner before deploying online.

The logged data in offline policy learning are generally used in two different ways: it can be used in Direct Reinforcement Learning, which trains the recommendation policy directly using the logged data (refers to as learning); or it can be used in Indirect Reinforcement Learning, which first builds a simulator to imitate customers’ behaviors and then learns the policy via querying the simulator (refers to as planning). Current methods of direct reinforcement learning include Monte Carlo (MC) and Temporal Difference (TD). Offline MC estimation with importance sampling guarantees an unbiased estimation, but it suffers from the problem of high variance. Especially in realistic recommender systems, there are often millions of candidate items for the recommendation, leading to an
extremely large action space and an unbounded importance weight of training samples. As a result, it requires both large training samples and computation resources to achieve statistical efficiency in training. TD-based methods improve efficiency by using the bootstrapping technique in estimation. However, it is confronted with another notorious problem called Deadly Triad, that is, the problem of instability and divergence arises whenever combining function approximation, bootstrapping and offline training [35] (see the example of training divergence in Sec 5.3.1 Figure 4). Unfortunately, in recommender systems, due to the complexity of modeling customer behaviors, most state-of-the-art methods [47, 48] that are designed with neural architectures, will encounter inevitably the \textit{Deadly Triad} problem in offline policy learning.

The indirect reinforcement learning approach using a simulator, theoretically, does not incur the real-world cost and can provide unlimited simulated experiences to learn the recommendation policy. However, building an effective recommendation simulator is in its own non-trivial problem, which has not been well explored up to now. Preliminary works on recommendation simulators [28, 47] typically ignore the selection bias of logged data [29], resulting in a biased simulator. Moreover, in those existing methods, a simulator was built before performing policy learning and is kept unchanged during policy learning, that is, a fixed simulator serves all intermediate policies when performing policy improvement. We believe that the simulator should also be updated constantly, in accordance with the improved target policies, to obtain the customized optimal accuracy of the simulation.

To address these issues, inspired by Dyna-Q [23, 34], we integrate learning (direct reinforcement learning) and planning (indirect reinforcement learning) in a unified framework, named \textit{Pseudo Dyna-Q (PDQ)}. Different from Dyna-Q, the recommendation policy is trained without the requirement of real customer interactions. Specifically, we introduce an environment model, referred to as \textit{world model}, to simulate the environments and generate simulated customer experiences in offline policy learning. The policy learning is then decomposed into two iterative steps: in the first step, the world model is constantly updated in accordance to current recommendation policy by de-biasing the selection bias with importance sampling; the second step improves the recommendation policy with Q-learning, via both the logged data and the world model (referred to as direct reinforcement learning and planning respectively). Compared with existing approaches [28, 47], the advantages of PDQ lie in two aspects: 1) PDQ breaks the Deadly Triad by employing a world model for planning; 2) the bias induced by simulator is minimized by constantly updating the world model and by a direct off-policy learning.

To these ends, our main contributions in this work are as follows:

- We present Pseudo Dyna-Q (PDQ) for interactive recommendation, which provides a general framework that can be instantiated in different neural architectures, and tailored for specific recommendation tasks.
- We conduct a general error analysis for the world model and show the connection of the error and dispersity between recommendation policy and logging policy.
- We implement a simple instantiation of PDQ, and demonstrate its effectiveness on two real-world large scale datasets, showing superior performance over the state-of-the-art methods in interactive recommendation.

\section{PRELIMINARIES}

\textbf{Online interactive recommendation.} In general, we assume a typical interactive recommendation setting between the customer and the recommender system – in each interaction, the customer is recommended an item $i_t \in I$ and provides a feedback $f_t \in \mathcal{F}$ (i.e., skipping, clicking or purchase) at the $t$-th interaction; then the system recommends new items $i_{t+1}$ for the customer until the customer leaves the platform. Here, $I$ is the set of candidate items for recommendation, $\mathcal{F}$ is the set of possible feedback. Such an interactive process can be formally formulated as a Markov Decision Process (MDP).

Given the observation on past interactions $s_t = \{u, i_1, f_1, \ldots, i_{t-1}, f_{t-1}\}$ (i.e., the state in MDP with $s_t \in S$), the recommender is modeled by a conditional distribution $\pi : S \times I \rightarrow \mathbb{R}$ with $\pi(i|s_t)$ (i.e., the policy in RL community) being probability of recommending item $i$ at the $t$-th interaction. The interaction between the customer and the recommender will generate a recommendation trajectory as $\xi = (s_0, i_0, r_0, \ldots, s_t, i_t, r_t, \ldots, s_T)$, where $r_t \in \mathbb{R}$ is a reward associated with customer’s feedback, e.g., a click or a purchase. The aim of a recommender agent is to learn a policy $\pi$ for maximizing the reward values of a trajectory $\xi$, formulated as:

$$\pi^* = \arg \max_{\pi \in \Pi} \eta(\pi).$$

Here, $\eta(\pi)$ is the expected discounted reward by following the policy $\pi$:

$$\eta(\pi) = \mathbb{E}_{\xi \sim P^\pi} \left[ \sum_{t=0}^{T} \gamma^t r(s_t, i_t) \right],$$

where $r(s_t, i_t) \in [0, r_{max}]$ is the reward associated with customer’s feedback; $\gamma \in (0, 1]$ is hyper-parameter for discounting the long-term rewards.

\textbf{Offline learning task.} As mentioned earlier, due to the high cost and risk of deploying an immature recommendation policy, the offline learning task aims to learn a recommendation policy using a large logged trajectories to avoid interactions with real customers online.

Given the logged trajectory data $\mathcal{D} = \{\xi^{(k)}\}_{k=1}^{N}$, where $\xi^{(k)} \sim P^\pi_0$, $N$ is the total number of trajectories in the logged data, the aim of the offline learning task is the same as that of the online recommender, as expressed in Equation 1. The difference is that the trajectories are supposed to be generated independently by a logging policy $\pi_0$, i.e., $\xi^{(k)} \sim P^\pi_0$; instead of being generated through interactions with real customers, i.e., $\xi^{(k)} \sim P^\pi_0$ as in Equation 2. The offline learning task needs to handle the policy bias to learn an optimal policy $\pi^*$ without interactions with real customers online.

\section{POLICY LEARNING FOR RECOMMENDER VIA PSEUDO DYNA-Q}

The proposed PDQ recommender agent is shown in Figure 1. It consists of two modules:
A world model can be learned effectively by minimizing the errors following logging policy \( \pi \). Here, the function is equivalent to imitate customers’ feedback. Formally, the goal of the world model is to imitate customers’ feedback, e.g., a click or a purchase, learning the reward function is equivalent to imitate customers’ feedback. Formally, the world model can be learned effectively by minimizing the errors between online and offline rewards:

\[
\ell(\pi; \theta_M) = \text{Err}(\eta(\pi), \eta(\pi; \theta_M)) \\
= \mathbb{E}_{\xi \sim p_\pi} \left[ \sum_{t=0}^{T-1} y^t \Delta_t(r_t, r_t(\theta_M)) \right],
\]

where \( \eta(\pi) \) is the expected discounted reward following policy \( \pi \) in real world, while \( \eta(\pi; \theta_M) \) is the expected reward of following \( \pi \) with \( \theta_M \) as the parameter in the world model. \( \Delta(\cdot) = \frac{\Delta(\cdot)}{K} - 1 \in [-1,0] \) measures the difference between the real reward \( r(\cdot) \) and learned reward \( r(\cdot; \theta_M) \) with \( \delta(\cdot; \theta_M) \in [0,K] \) as a scalar loss function.

Since \( \xi \) in the dataset \( D \) is generated by logging policy \( \pi_b \), the objective function can be rewritten as:

\[
\theta^*_M = \arg\min_{\theta_M \in \Theta} \mathbb{E}_{\xi \sim p_\pi} \left[ \sum_{t=0}^{T-1} y^t \Delta_t(r_t, r_t(\theta_M)) \right].
\]

Here, \( \Delta_t(\theta_M) \) is the shorthand for \( \Delta_t(r_t, r_t(\theta_M)) \), \( \xi \) is generated by following logging policy \( \pi_b \), \( \rho_0 \) is the importance ratio to correct the discrepancy between recommendation policy and logging policy. Accordingly, \( \hat{\theta}^*_M \) can be acquired by solving the sample average approximation:

\[
\hat{\ell}(\pi; \theta_M) = \frac{1}{N} \sum_{k=1}^{N} \omega_b(k) \Delta_t(\theta_M).
\]

However, this estimator has unbounded variance, since \( \omega_b(k) \) can be arbitrarily big when \( \pi_b \approx 0 \), which causes \( \hat{\ell}(\pi; \theta_M) \) to be far away from the true risk \( \ell(\pi; \theta_M) \). This problem can be fixed by “clipping” the importance sampling weights [16] as:

\[
\hat{\ell}^c(\pi; \theta_M) = \frac{1}{N} \sum_{k=1}^{N} \min\left(\omega_b(k), c\right) \Delta_t(\theta_M).
\]

where \( c > 0 \) is a hyper-parameter chosen to balance the bias and variance in the estimator, i.e., a smaller value of \( c \) means toleration of a larger bias in the estimator.

3.1.2 The Error Bound and Its Induced Regularizer. The variance of \( \hat{\ell}^c(\pi; \theta_M) \) in Equation (6) varies very differently across different history. Consider two policies \( \pi_1 \) and \( \pi_2 \), where \( \pi_1 \) is similar to \( \pi_b \), but \( \pi_2 \) is not. Importance sampling gives us lower variance estimates for \( \hat{\ell}^c(\pi_1; \theta_M) \), but higher variance estimates for \( \hat{\ell}^c(\pi_2; \theta_M) \). Following the intuition above, we can get the upper bound of the error function in Equation (7) (the proof is provided in Appendix).

\[
\text{Theorem 3.1.} \quad \text{Let} \quad \rho^s_{n}(s,i) \quad \text{be the probability of arriving at state} \quad s \quad \text{and taking the action} \quad i \quad \text{at} \quad \text{timestep} \quad t \quad \text{when policy} \quad \pi. \quad \text{Define the divergence between the policy} \quad \pi \quad \text{and} \quad \pi_b, \quad \text{as} \quad D_f(\pi \| \pi_b) = \sum_i y_t d_f(\rho^t_{n}(s,i)) = \sum_i y_t \left( \sum_{(s,i)} f(\rho^t_{n}(s,i)) \right), \quad \text{where} \quad d_f(\cdot) \quad \text{is the f-divergence with} \quad f(x) = x^2 - x. \quad \text{With probability at least} \quad 1 - \epsilon, \quad \text{for all} \theta_M, \quad \text{we have} \quad \ell(\pi; \theta_M) \leq B, \quad \text{where}
\]

\[
B = \hat{\ell}^c(\pi; \theta_M) + \sqrt{\frac{18(1-\gamma)D_f(\pi \| \pi_b) + 1)Q_{\theta_M}(n, \xi)}{nT}}
\]

\[
+ \frac{c \cdot 45Q_{\theta_M}(n, \xi)}{n - 1},
\]

and \( Q_{\theta_M}(n, \xi) \) measures the capacity of the reward function family.

The second part of the error bound in Equation (7) indicates that, in order to tighten the upper error bound, we can force \( \rho^s_{n} \) and \( \rho^s_{\pi_b} \) to be as close as possible. We are here unable to intervene \( \rho^s_{\pi} \) directly (which is typically obtained through maximizing \( \eta(\pi; \theta_M) \) in policy learning phase, see in Sec 3.2). Alternatively, we can intervene the reward function \( r(\theta_M) \) to force \( \rho^s_{\pi_b} \) to approach \( \rho^s_{\pi_b} \), as long as we can find out the relationship among \( r(\theta_M) \), \( \rho^s_{\pi} \) and \( \rho^s_{\pi_b} \). The relationship is stated in Lemma 3.2 (the proof is provided in Appendix).

\[
\text{Lemma 3.2.} \quad \text{Assuming that} \quad r^* \quad \text{is the reward function which satisfies} \quad r^*(s_t, i_t) = \rho^s_{\pi_b}(s_t, i_t), \quad \pi^* \quad \text{is the optimal policy maximizing} \quad \eta(\pi) \quad \text{under} \quad r^*. \quad \text{For any indicator policy} \quad \pi, \quad \text{we have} \quad D_f(\pi \| \pi_b) \geq D_f(\pi^* \| \pi_b). \quad \text{As stated in Lemma 3.2, the optimal policy} \quad \pi^* \quad \text{for} \quad r(s_t, i_t; \theta_M) \quad \text{has minimal divergences} \quad D_f(\pi \| \pi_b) \quad \text{when the reward} \quad r(s_t, i_t; \theta_M) \quad \text{is proportional to} \rho^s_{\pi_b}(s_t, i_t). \quad \text{In other words, the state action pair} \quad (s_t, i_t) \quad \text{with high visiting frequency} \quad \rho^s_{\pi_b} \quad \text{should be assigned higher reward by} \quad r(s_t, i_t; \theta_M). \quad \text{This inspires us to add a regularizer intervening the reward function} \quad r \quad \text{to force} \quad \rho^s_{\pi_b} \quad \text{to approach} \rho^s_{\pi_b}. \quad \text{This}
\]
The offline (s_t, i_t) are generated by following π_k, which follows the probability of ρ_k^θ(s_t, i_t). The regularized error function is:

$$E_{\pi} [\sum_{T=0}^{T-1} \min_{c=0,1} \eta \Delta_t(\theta_M)]$$ (8)

where Δ(θ_max, r_t(θ_M)) is the regularizer for encouraging π to visit (s_t, i_t) (heuristically reducing the divergence D_f(π||π_k) in Theorem 3.1). λ is the hyper-parameter that controls the influence of the regularizer term. Accordingly, $\theta_M$ can be obtained by solving the sample average approximation

$$E_{\pi} [\sum_{T=0}^{T-1} \min_{c=0,1} \eta \Delta_t(\theta_M)]$$ (9)

3.2 Policy Learning

We use Q-Learning [22] to improve the recommendation policy via using the experiences from the world model and via directly using the logged experiences. In each time-step t of recommendation, the recommender agent observes the state of customer s_t, and chooses the item i_t to recommend using an ε-greedy policy (i.e., with probability 1 − ε selecting the max Q-value action, with probability ε randomly choosing an action) w.r.t. the approximated value function $Q(s_t, i_t)$, which can be customized for specific recommendation tasks. The agent then receives the response $r(s_t, i_t; \theta_M)$ from world model and updates the state to $s_{t+1}$. Finally, we store the experience $(s_t, i_t, r_t, s_{t+1})$ in a large replay buffer $M$ from which samples are taken in mini-batch training. The cycle continues until the customer leaves the platform.

We improve the value function $Q(s_t, i_t; \theta_Q)$ by adjusting $\theta_Q$ to minimize the mean-square loss function, defined as follows:

$$\ell(\theta_Q) = E_{(s_t, i_t, r_t, s_{t+1}) \sim M} \left[ (y_t - Q(s_t, i_t; \theta_Q))^2 \right]$$ (10)

where $y_t$ is the target value based on the optimal Bellman Equation [35]. By differentiating the loss function w.r.t. $\theta_Q$, we arrive at the following gradient:

$$\nabla_{\theta_Q} \ell(\theta_Q) = E_{(s_t, i_t, r_t, s_{t+1}) \sim M} \left[ (r_t + \max_{i_{t+1}} Q(s_{t+1}, i_{t+1}; \theta_Q) - Q(s_t, i_t; \theta_Q)) \nabla_{\theta_Q} Q(s_t, i_t; \theta_Q) \right]$$ (11)

In fact, the policy learning maximizes the $\eta(\pi; \theta_M)$ of the world model, where the regularization in Equation (9) is actually conveyed in $\theta_Q$.

Finally, we implement an interactive training procedure, as shown in Algorithm 1, where we specify the order in which they occur within each iteration.

4 AN INSTANTIATION OF PSEUDO DYNA-Q

We have described a general framework of PDQ. In this section, we present a simple instantiation of the framework. Note that the proposed framework is not limited to the instantiation we describe here. More sophisticated designs of state representation, world model and Q-value network could be used, according to the specific recommendation tasks. As shown in Figure 2, the instantiation of Pseudo Dyna-Q contains three parts: (a) The state tracker for tracking current customer’s preferences, e.g. encoding both the long-term and temporary interests into a dense state representation $s_t$; (b) The Q-Value Network for predicting the Q-value of the policy; (c) The world model for generating pseudo customer’s feedback.

4.1 State Tracker

RNN is often used to keep track of the states. In reality, customers’ current interests are often related to the earlier items in addition to the recent ones, and RNN-based methods are unable to cope with such long term dependency. Recent works on memory network and self attention show effectiveness on this issue [20, 33, 41], and we here adopt such a design. Given the observation $s_t = \{u_t, i_1, f_1, \ldots, i_{t-1}, f_{t-1}\}$, the entire set of $\{i_j\}$ are converted into embedding vectors $\{\mathbf{f}_j\}$ of dimension $H$ by embedding each $i_j$ in a continuous space, which, in the simplest case, is an embedding matrix $A$ (of size $I \times H$). To represent the feedback information

**Algorithm 1: The training of Pseudo Dyna-Q.**

Input: $D, \epsilon, L, K$

Output: $M(s, i; \theta_M), Q(s, i; \theta_Q)$

1. Randomly initialize parameters $\theta_Q, \theta_M \leftarrow \text{Uniform}(-0.1, 0.1)$

2. # Pretraining the world model.

3. for $j = 1 : K$ do

4. Sample random mini-batches of $(s_t, i_t, r_t, s_{t+1})$ from $D$

5. Set $f_j$ according to $r_t$

6. Set $e_j$ according to $s_{t+1}$

7. Update $\theta_M$ via mini-batch SGD w.r.t. the loss in Equation (9)

9. # Iterative training of world model and Q-value network.

10. repeat

11. for $j = 1 : N$ do

12. # Sampling training data by querying the world model.

13. $e = \text{False}$

14. sample a initial customer $u$ from customer set; initialize $s = \{u\}$

15. while $e$ is false do

16. sample a recommendation $i$ by $e$-greedy w.r.t Q-value;

17. execute $i$;

18. world model responds with $f, e$;

19. set $r$ according to $f$;

20. store $(s, i, r, s')$ in buffer $M$;

21. update $s \leftarrow s'$;

22. end

23. # Adding logged data to the training samples.

24. Sampling $(s, i, r, s')$ from $D$ and storing in buffer $M$;

25. # Updating the Q-value network.

26. for $j = 1 : L$ do

27. Sample random mini-batches of $(s_t, i_t, r_t, s_{t+1})$ from $M$

28. Update $\theta_Q$ via mini-batch SGD w.r.t. Equation (11)

29. end

30. # Updating the world model.

31. for $j = 1 : K$ do

32. Sample mini-batches of $(s_t, i_t, r_t, s_{t+1})$ from $D$

33. Set $f_j, e_j$ according to $r_t, s_{t+1}$

34. Update $\theta_M$ via mini-batch SGD w.r.t. the loss in Equation (9)

35. end

36. end

37. until convergence;
for Q
feedbacks
where
where Softmax
The approximation of Q-value is accomplished by the inner prod-
to the Q-value by the inner product between tracked cus-
Finally, the state
\[ t_f, j = F_f i_j, \]
where \( F_f \in \mathbb{R}^{H \times H} \) is a projection matrix for a specific feedback \( f_f \). In the embedding space, we compute the match score between the recommendation embedding \( i_j \) and each memory cell \( i_f, j \) by taking the inner product followed by a softmax:
\[ \alpha_j = \text{Softmax}(i_j^T t_f, j), \]
where \( \text{Softmax}(z) = \frac{\exp(z_j)}{\sum_{j} \exp(z_j)} \). \( i_j \) is the embedding of recommendation. Defined in this way, \( \alpha \) is a probability vector over the inputs. Finally, the state \( s_t \) is formed by concatenating customers’ embedding \( u \in \mathbb{R}^U \) and \( \alpha \) weighted sum of inputs as:
\[ s_t = \left[ u \sum_{j=0}^{t-1} \alpha_j i_f, j \right], \]
where \([ \cdot, \cdot] \) means concatenation operation. Given it, the contextual feature for decision making is formulated as:
\[ \phi(s_t, i_t) = W_c [s_t, i_t] + b_c, \]
where \( W_c \) and \( b_c \) are the weight and bias terms.

4.2 The Q-value Network
The approximation of Q-value is accomplished by the inner products of the dense state embedding with a weight vector as follows:
\[ Q(s_t, i_t; \theta_Q) = w_q^T \phi(s_t, i_t) + b_q. \]

5 EXPERIMENTS
In this section, we perform empirical evaluations of our proposed PDQ on two large collections of real-world customer logs extracted from e-commerce platforms. The source code can be found at Github: https://github.com/zoulixin93/pseudo_dyna_q.

5.1 Experimental Settings
Dataset. We adopt the following two public datasets in our experiments.
- Taobao\(^4\): Taobao is the largest E-commerce platform in China. Taobao dataset contains a subset of customer behaviors including click, purchase, adding item to shopping cart and item favoring from November 25, 2017 to December 03, 2017.
- Retailrocket\(^5\): Retailrocket is a dataset collected from a real-world e-commerce website over a period of 4.5 months, which contains customers’ behaviour data (i.e. events like “clicks”, “add to carts” and “transactions”).

Detailed statistic information, including the number of customers, items and behaviors, of these datasets is given in Table 1.

Baseline. We compare our model with the state-of-the-art baselines, including both supervised learning based methods and reinforcement learning based methods.

\(^4\)https://tianchi.aliyun.com/datalab/dataSet.html?dataId=649
\(^5\)https://www.kaggle.com/retailrocket/e-commerce-dataset/home

Table 1: Statistics of the datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Customers</th>
<th>#Items</th>
<th>Total Behaviors</th>
<th>Behaviors per Customer</th>
<th>Behaviors per Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taobao</td>
<td>986,240</td>
<td>4,161,799</td>
<td>100,144,665</td>
<td>101.5419</td>
<td>24.0628</td>
</tr>
<tr>
<td>Retailrocket</td>
<td>81,620</td>
<td>103,873</td>
<td>948,537</td>
<td>11.6214</td>
<td>9.1317</td>
</tr>
</tbody>
</table>

Here, \( w_q \) and \( b_q \) are the weight vector and bias terms. The update of Q-value network follows the Equation (11).
• BPR [26]: It optimizes the matrix factorization model with a pairwise ranking loss. This is a popular method for item recommendation. However, it ignores the sequential information of recommendation and cannot optimize the long-term reward in recommendation.
• FPMC [27]: It learns a transition matrix based on underlying Markov chains. Sequential behaviors are modeled only between the adjacent transactions.
• GRU4Rec [17]: This is a representative approach that utilizes RNN to learn the dynamic representation of customers and items in recommender systems.
• NARM [20]: This is a state-of-the-art approach in personalized trajectory-based recommendation with RNN models. It uses attention mechanisms to determine the relatedness of the past purchases in the trajectory for the next purchase.
• DQN-R [48]: It is an elegant and concise off-policy reinforcement learning method, which has been employed for sequential e-commerce recommendation in [48].
• DDPG-KNN [11]: DDPG is an actor-critic, model-free framework. In [11], it has been adapted for discrete recommendation by combining DDPG with an approximate KNN method.
• PDQ: Our PDQ model that utilizes a world model to imitate customer’s feedback and learns an offline policy by combining planning and direct RL. To verify the effect of different components, we also test the following degenerated PDQ models:
  - PDQ(N): The naive PDQ, which separately optimizes the world model and recommendation policy and does not handle the distribution mismatch between the logging policy and recommendation policy.
  - PDQ(IM): It iteratively trains the world model and offline policy, and employs clipping importance sampling to deal with the mismatch between logging policy and recommendation policy.
  - PDQ(IM+R): Our integrated PDQ(IM+R) model, which regularizes the generalization error by minimizing the distribution divergence between the logging policy and recommendation policy.

Parameter Setting. The state tracker has one hidden layer and 200, 100 hidden units for Taobao and Retailrocket respectively. All the baseline models share the same layer and hidden nodes configuration for the neural networks. e-greedy is always applied for exploration but discounted with increasing training epoch. The value γ for clipping importance sampling is set to 5. We set the discount factor γ = 0.9. The buffer size of M is set as 10000. The target value function is updated at the end of each epoch. In each epoch, the mini-batch size is 256. The networks are trained with SGD [3] with a learning rate of 0.005. Unless otherwise specified, the hyper-parameter λ for regularisation is 0.01. We used TensorFlow to implement the pipelines and trained networks with an Nvidia GTX 1080 ti GPU cards. All the experiments are obtained by an average of 5 repeat runs.

5.2 Online Testing Experiments

5.2.1 Simulation Setting. To perform evaluation of RL methods on ground-truth, a straightforward way is to collect a large logged dataset and evaluate the learned policy through online A/B test, which, however, could be too expensive and commercially risky for the platform. Similar to [5, 11, 28], we demonstrate how the proposed method would perform on a real world recommender system by constructing a simulated customer model utilizing data from Taobao and Retailrocket. Without loss of generality, we regard the “clicks”, “add to carts”, “transactions” as positive feedback (clicks) and assume a standard rank-H-restricted matrix factorization model [26] Pr(click(u, i)) = Sigmoid([u, i]H) for customers’ clicks, where [u, i] ∈ RH are the latent factors learned by fitting Taobao and Retailrocket dataset through BPR-MF. The ranking H for Taobao and Retailrocket are set as [100, 50], learning rate is [0.1, 0.1], and the maximum iteration is 2000 before convergence.

Apart from the feedback, we need to simulate customers’ patience on the platform – when to end the trajectory after losing patience. Similar to [11], we assume that the ending probability is correlated with customers’ feedback. In other words, if the presented item is accepted then the trajectory has a small ending probability; if the item is not accepted then the trajectory has a higher ending probability. However, this assumption is not good enough for building a real customer simulator, since under this assumption, the learned policy could repeatedly recommend similar items, which may annoy customers. For this reason, we further assume that the ending probability is also related to the diversity of recommended items (i.e. diverse recommendations are more attractive to customers [9]). Formally, based on the intuitions, the ending probability for a trajectory is set as:

\[
Pr(\text{ending}|u, i_0, i_1, \ldots, i_t) = 1 - \frac{2}{H} \sum_{m,k \in \{1, 2, \ldots, t\}} \text{entropy}(i_m, i_k),
\]

where \(\text{entropy}(i_m, i_k) = \sum_{j=1}^{H} i_{m,j} \log \frac{i_{m,j}}{i_{k,j}}\) measures the distance between item \(m\) and item \(k\).

5.2.2 Evaluation Setting. The principle of our evaluation is to recover the real offline learning and recommendation scenario. To this end, we first sample 9.8M and 891k trajectories as the training dataset from the simulators built using Taobao and Retailrocket datasets. Here, the logging policy \(\pi_b\) is set as \(\pi_b(i|i) = \text{Softmax}(\frac{u^T_i}{\tau})\), where \(\tau\) is the temperature to control the performance of logging policy. In real learning tasks, \(\pi_b\) should not be too random because online platforms usually spend a lot of effort in designing their recommender systems to satisfy customers’ demands. Therefore, we set the \(\tau = 5\), to ensure a medium level logging policy. After that, we train the model using the offline trajectories and evaluate the learned policies on the simulator.

5.2.3 Main Results. Results by an average of 5 repetitive experiment runs are obtained and we report the three metrics in Table 2 and Figure 3. For each recommendation agent, we report its results in terms of the average clicks per trajectory (Clicks), average diversity of recommendations (Diversity) and the average number of interactions (Horizons) [11, 23], which measure the goodness of the whole trajectory recommendations. Figure 3 shows the learning curves on average clicks per session of different kinds of recommendation agents on these two datasets. From Table 2 and Figure 3, we have following observations:
Table 2: Performance comparison of different recommendation agents on offline learning tasks.

<table>
<thead>
<tr>
<th>Agents</th>
<th>Taobao</th>
<th>Retailrocket</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Epoch=1000</td>
<td>Diversity</td>
</tr>
<tr>
<td>DQN-R</td>
<td>0.8624</td>
<td>0.0157</td>
</tr>
<tr>
<td>PDQ (1M)</td>
<td>1.8862</td>
<td>0.0043</td>
</tr>
<tr>
<td>PDQ (2M)</td>
<td>1.9926</td>
<td>0.0747</td>
</tr>
<tr>
<td>PDQ (2M) (R)</td>
<td>1.8361</td>
<td>0.0610</td>
</tr>
<tr>
<td>PDQ (3M) (R)</td>
<td>1.8755</td>
<td>0.0006</td>
</tr>
</tbody>
</table>

* indicates the statistically significant improvements (i.e. two-sided t-test with p < 0.01) over the best baseline.

Figure 3: Learning curves of PDQ agents and baseline models.

5.3 Analysis

In this section, we further analyze the effectiveness of the proposed framework for the offline recommendation task based on the two datasets.

5.3.1 The Deadly Triad in Recommendation.

As previously mentioned, directly deploying TD based methods may not be safe in offline training tasks (i.e. the headache of Deadly Triad\(^3\)). To study the problem of Deadly Triad in the recommendation, the average Q-value over training epochs (averaged over 5 repeat experiments) has been shown in Figure 4. We can see that: (1) The Q-value of DQN-R and DDPG-KNN (0.1N) are not stable in offline training. The Q-value of DQN-R quickly blows up after several iterations. DDPG-KNN’s Q-value seems stable in the beginning but it also changes to infinity at a certain point. This phenomenon may be caused by the difference in estimating the next state value between these two methods. In DQN-R, the maximal next Q-value \(Q^\pi(s_t, a_t)\) is chosen for TD estimation. However, in DDPG-KNN, the sampling action value \(Q^\pi(s_t, a_t)\) is used. The max operation accelerates the blow-up of Q-value. (2) Compared with directly learning an offline policy, PDQ can effectively solve this problem by building a world model bridging the gap between offline and online training. In Figure 4, the Q-value of PDQ converges to a stable value at the end.
5.3.2 Influence of Regularizer Weight. We investigate how the performance varies w.r.t. the regularizer weight $\lambda$ in Equation (8). The Clicks, Diversity, and Horizons are shown in Figure 5 with different $\lambda$ ranging from 0.001 to 1. The experimental results demonstrate that if the regularizer is too large, it will hurt performance. On the contrary, a too-small regularizer will have a limited influence on the variance of the learned world model, which is consistent with our intuition about the regularizer.

5.3.3 The Effectiveness of State Tracker. We propose a memory-based neural architecture to track customers' interests in the interactive recommendation. To verify its effectiveness, we train two online DQN agents on the simulator with different state tracker: one with our proposed memory-based state tracker, named DQN(MEM); The other DQN(GRU) employs GRU as function approximation, which has been widely used in recommendation tasks [17, 20]. Figure 6 presents the learning curve on two datasets. DQN(MEM) performs better than DQN(GRU), which suggests that the proposed memory and self-attention based state tracker is more capable of modeling complex interactions than GRU as function approximation.

6 RELATED WORK
Recommender systems have attracted a lot of attentions from the research community and industry. Being supervised by the history records is the common practice in majority models, including traditional factorization methods [4, 14, 19, 25], deep neural models, such as multilayer perceptron [8], denoising auto-encoders [44], convolutional neural network (CNN) [2, 38], recurrent neural network (RNN) [13, 15, 20, 43], memory network [7] and attention architectures [1, 6]. Based on the partial observed history dataset, these existing models usually predict a customer's feedback by a learning function to maximize some well-defined evaluation metrics in ranking, such as Recall, Precision and NDCG [10]. However, most of them are myopic because the learned policies are greedy with estimating customers' feedback and unable to optimize customers’ feedback in the long run.

Recently, reinforcement learning-based approaches have attracted a lot of attention in recommender systems. The core idea of RL models is learning an effective policy to maximize the expected reward in the long run. The most common approach is learning the policy by learning empirical rewards from interaction with real customers, e.g. contextual bandit (i.e. 1-horizon MDP) based recommender methods [21, 24, 42, 45], Markov Decision Process (MDP) based recommendation methods [5, 11, 46–51]. Contextual bandit models handle the notorious explore/exploit dilemma in online environment for the cold start problem; while MDP based methods design different neural network architectures to extract interactive information from customer status. Due to the fact that learning a policy online by interacting with real customers may lead to poor customer experiences [37]. Hence the most common way is to utilize the history data to train an offline policy, which enables the recommender system to get past its blundering stage in an offline environment without putting anyone in an unfriendly experience.

The offline policy learning, which is a tempting challenge, has attracted great interest in RL community to design stable and efficient learning algorithms. Algorithms in existing models can be classified into Monte Carlo (MC) and temporal-difference (TD) methods. Due to the efficiency problem of MC, its usage has been limited to off-policy evaluation [12, 18, 39]. As for TD, it has a black cloud, i.e. deadly triad³, hanging over its head. Current solutions for Deadly Triad are limited to linear function approximation, such as GTD2 [36] and GReTrace($\lambda$) [40]. However, none of them can be applied to complex function approximation, i.e. neural networks.

7 CONCLUSION
In this work, we investigated offline policy learning in recommender systems, which is usually more practical than online policy learning in practice. An offline policy learning strategy—Pesudo Dyna-Q (PDQ) was proposed for interactive recommendation. PDQ performs offline policy learning through both model-based indirect and direct offline learning, where the world model is introduced to simulate the environments and assist TD-based policy improvement. We also provided a general error analysis of the world model’s risk function, and based on the analysis, the world model is designed to keep adaptively optimized for specific recommendation policies.
during policy learning. The TD based Q-Learning in offline setting is hence able to prevent from instability of convergence, and perform policy improvement effectively, via both logged experiences and querying the simulator. Extensive experiments on two real world large scale datasets showed that the instantiated PDQ based on neural networks outperforms state-of-the-art methods noticeably.

REFERENCES