Off-Policy Recommendation System Without Exploration

Chengwei Wang[1,3], Tengfei Zhou[3], Chen Chen[1,3](✉), Tianlei Hu[1,3], and Gang Chen[2,3]

[1] The Key Laboratory of Big Data Intelligent Computing of Zhejiang Province, Zhejiang University, Hangzhou, China
{rr,cc33,htl}@zju.edu.cn

[2] CAD and CG State Key Lab, Zhejiang University, Hangzhou, China
cg@zju.edu.cn

[3] College of Computer Science and Technology, Zhejiang University, Hangzhou, China
zhoutengfei@zju.edu.cn

Abstract. Recommendation System (RS) can be treated as an intelligent agent which aims to generate policy maximizing customers' long term satisfaction. Off-policy reinforcement learning methods based on Q-learning and actor-critic methods are commonly used to train RS. Though these methods can leverage previously collected dataset for sampling efficient training, they are sensitive to the distribution of off-policy data and make limited progress unless more on-policy data are collected. However, allowing a badly-trained RS to interact with customers can result in unpredictable loss. Therefore, it is highly desirable that the off-policy method can stably train an RS when the off-policy data is fixed and there is no further interaction with the environment. To fulfill these requirements, we devise a novel method name Generator Constrained Q-learning (GCQ). GCQ additionally trains an action generator via supervised learning. The generator is used to mimic data distribution and stabilize the performance of recommendation policy. Empirical studies show that the proposed method outperforms state-of-the-art techniques on both offline and simulated online environments.

1 Introduction

Recommender System (RS) is one of the most important applications in artificial intelligence [15,20]. An intelligent RS can significantly reduce users' searching time, greatly enhance their shopping experience and bring considerable profits to vendors.

From the Reinforcement Learning (RL) perspective, RS is an autonomous agent that intelligently learns the optimal recommendation behavior over time to maximize each user's long term satisfaction through interacting with its environment. This offers us the opportunity to solve the recommendation task on top of

Supported by the National Key R&D Program of China (No. 2017YFB1201001).

the recent RL advancement. Considering that a previously collected customers' feedback dataset is often available for recommendation tasks, many researchers adopt the off-policy RL methods to extract patterns from the data [4,21,23].

Off-policy RL algorithms are often expected to fully exploit off-policy datasets. Nevertheless, these methods can break down when the datasets are not collected by learning agents. Theoretically, [2] points out that Bellman updates could diverge with off-policy data. The divergence issue would surely invalidate the performance of DQN agents. [12,16] find that in off-policy learning, the fix-point of Bellman updates may have poor quality even if the update converges. Empirically, [9] shows that off-policy agents perform dramatically worse than the behavioral agent when trained by the same numerical algorithm on the same dataset. Moreover, many researchers observe that these methods can still fail to learn the optimal strategy even when training data are deliberately selected by effective experts. All these observations suggest that off-policy methods are unstable to static datasets.

The instability of off-policy methods is highly undesirable in training an RS. One would hope that the RS has learned sound policies before deploying into a production environment. If its performance turns out to be unpredictable, deploying the RS would be risky. To stabilize off-policy methods, one can compensate for the performance of the RS by online feedbacks. That is, allow the off-policy agent to interact with customers and use the customers' feedbacks to stabilize its performance. In practice, collecting user's feedback is time-consuming, and deploying an unstable RS to interact with customers would greatly reduce their satisfaction. As a result, designing a stable off-policy RL method for RS which has reasonable performance for any static training set without further exploration, is a fundamental problem.

As indicated in [9,14], the instability issue of off-policy methods results from *exploration error* which is a fundamental problem with off-policy reinforcement learning. exploration error usually behaves as the value function is erroneously estimated on unseen state-action pairs. The exploration error can be unboundedly large, even if the value function can be perfectly approximated [9]. Moreover, it can accumulate during the training iterations [14]. It may misguide the training agent and make the agent take over-optimistic or over-pessimistic decisions. As a result, the training process becomes unstable and potentially diverging unless new data is collected to remedy those errors.

In this paper, we propose a novel off-policy RL method for RS to diminish the exploration error. Our method can learn recommendation policy successfully from large static datasets without further interacting with the environment. exploration error results from a mismatch in the distribution of data induced by the recommendation policy and the distribution of customers' feedback contained in the training data [9]. The proposed Generator Constrained deep Q-learning (GCQ) utilizes a neural generator to simulate customers' possible feedbacks. This generative model is combined with a Q-network which select the highest valued action to form recommendation policy. Furthermore, to reduce the decision time, we design the generator's architecture based on Huffman Tree. We show that with the generator pruning unlikely actions, the decision complexity can be reduced to $O(\log |A|)$ where $|A|$ is the number of actions, namely the number of items.

2 Off-Policy Recommendation Problem

A typical recommendation process can be formulated as a Markov Decision Process (MDP) $(\mathcal{S}, \mathcal{A}, r, P, \gamma)$ which is defined as follows.

- **State space** $\mathcal{S}$: The state $s_t^u = \{u, i_1, \ldots i_{c_t}\}$ contains the active user u and his/her chronological clicked items.
- **Action space** $\mathcal{A}$: The action space is the item set.
- **Reward** $r(s^u, a^u)$: Reward is the immediate gain of the RS after action a^u.

$$r(s^u, a^u) = \begin{cases} 1 & \text{if user } u \text{ clicks item } a^u \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

- **Transition probability** $P(s_{t+1}^u | s_t^u, a_t^u)$: The state transits as follows.

$$s_{t+1}^u = \begin{cases} s_t^u \cup \{a_t^u\} & \text{if user } u \text{ clicks item } a_t^u \\ s_t^u & \text{otherwise} \end{cases}$$

- **Discount rate** γ: $\gamma \in [0, 1]$ is a hyperparameter. It is the tradeoff between the immediate reward and long term benefits.

The off-policy recommendation problem can be formulated as follows. Let $B = \{(s_t^u, a_t^u, s_{t+1}^u, r_t^u)\}$ be the dataset collected by a unknown *behavior policy*. Construct a recommendation policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ such that the accumulated reward is maximized. For notation simplicity, we may omit the superscript of s^u, r^u, a^u in the following section.

3 Preliminaries

3.1 Q-Learning

Q-learning learns the state-action Q-function $Q(s, a)$, which is the optimal expected cumulative reward when the RS starts in state s and takes action a. The optimal policy π can be recovered from the Q-function by choosing the maximizing action that is $\pi(s) = \arg\max_{a \in \mathcal{A}} Q(s, a)$. The Q-function is a fix point of the following Bellman iteration:

$$Q^{k+1}(s_t, a_t) = r_t + \gamma \max_a Q^k(s_{t+1}, a). \tag{2}$$

with (s_t, a_t, s_{t+1}, r_t) sampled from B. The above update formula is called Q-learning in reinforcement learning literature. According to [9,14], Q-learning may have unrealistic value on unobserved state-action pairs, which results in large exploration error and makes the performance of an RS unstable.

3.2 Batch Constrained Q-Learning

To cope with the exploration error, [14] proposes the Batch Constrained Q-Learning (BCQ) method. BCQ avoids exploration error by explicitly constraining an agent's candidate actions in the training set. Specifically, BCQ estimates the Q-function by the following batch constrained Bellman update.

$$Q^{k+1}(s_t, a_t) = r_t + \gamma \max_{(s_{t+1},a)\in B} Q^k(s_{t+1}, a). \tag{3}$$

where "$(s_{t+1}, a) \in B$" means that there exist state s' and reward r' such that $(s_{t+1}, a, s', r') \in B$. Due to the sparsity of recommendation dataset, for most observed state s, there exists at most one action a such that $(s, a) \in B$. Thus, for most state-action pairs, the BCQ update (3) can be simplified to the following iteration

$$Q^{k+1}(s_t, a_t) = r_t + \gamma Q^k(s_{t+1}, a_{t+1}). \tag{4}$$

Such iteration implicitly assumes that the observed action a_{t+1} is optimal for state s_{t+1}, which is unrealistic because users' feedbacks are noisy.

4 Methodology

4.1 Generator Constrained Q-Learning

To prevent BCQ from overfitting into noisy data, we propose a new off-policy RL algorithm named Generator Constrained Q-learning (GCQ). GCQ utilizes a neural generator to recover the distribution of observed dataset. Then, the Q-function is updated on a candidate set sampled from the generator. Specifically, the main iteration of GCQ can be formulated as follows.

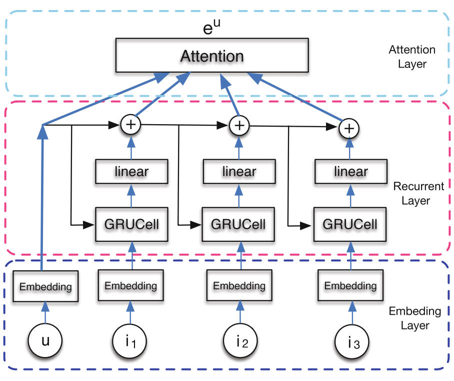
$$\begin{cases} A^k = \{a_i \sim g_{\theta_k}(a|s_{t+1})\}_{i=1}^{c} \\ Q^{k+1}(s_t, a_t) = r_t + \gamma \max\{Q^k(s_{t+1}, a)|a \in A^k\}. \end{cases} \tag{5}$$

where (s_t, a_t, s_{t+1}, r_t) is a randomly sampled tuple from B and $g_\theta(\cdot|s)$ is a neural generator which gives the conditional probability of actions. The size of candidate set c is a hyperparameter of GCQ method. When c is fixed to n, the number of items, GCQ becomes Q-Learning method.

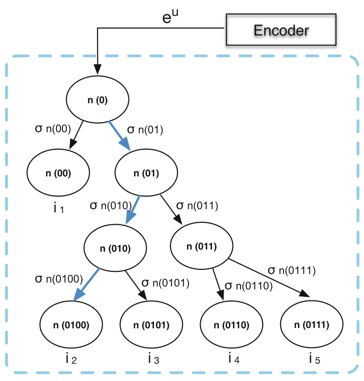
Since the state space of RS is large, it is impossible to compute the Q-function of each state-action pairs. To handle the difficulty, we approximate the unknown Q-function by a deep neural network $Q_\theta(s, a)$ a.k.a deep Q-net where θ is its parameter.

4.2 Architecture of State Encoder

Obviously, both Q-net and generator need an encoder to extract features from a state $s = \{u, i_1, \ldots i_T\}$. According to [3], a shared encoder generalizes better than multiple task-specified encoders. Therefore, we use the same encoder for Q-net $Q_\theta(s, \cdot)$ and generator $g_\theta(\cdot|s)$. We depict the structure of encoder in Fig. 1(a).



(a) Encoder



(b) Generator

Fig. 1. Neural architectures of proposed networks

Embedding Layer. The embedding layer maps a user or an item to correspondent semantic vector. Formally, Let $\mathbf{U} \in \mathbb{R}^{m\times d}$ and $\mathbf{V} \in \mathbb{R}^{n\times d}$ be the embedding matrix of user and item respectively. The embedding vector of user u and item i can be expressed as follows.

$$p_u = \mathbf{U}[u], \quad q_i = \mathbf{V}[i] \tag{6}$$

where we use $\mathbf{X}[k]$ to denote k-th row of matrix $\mathbf{X}$.

Residual Recurrent Layer. The layer transforms the sequence $s^u = \{u, i_1, \ldots i_T\}$ into hidden states. In the field of sequence modeling, GRU [6] and LSTM [10] are arguably the most powerful tools. However, both recurrent structures suffer from gradient vanishing/exploding issues for long sequences. Inspired by that residual network has stable gradients [19], we proposed a variant of GRU cell with residual structure. Specifically, we use the following recurrent to map the state s into hidden states $\{h_t\}_{t=0}^{t=T}$.

$$h_t = \begin{cases} p_u & \text{if}\, t = 0 \\ h_{t-1} + \mathbf{W} \cdot \text{GRUCell}(h_{t-1}, q_{i_t}) & \text{otherwise} \end{cases} \tag{7}$$

where p_u is the embedding vector of user u, q_{i_t} is the embedding vector of item i_t, and $\mathbf{W}$ is an alignment matrix.

Fast Attention Layer. The layer utilizes attention mechanism to aggregate hidden states $\{h_t\}_{t=0}^{t=T}$ into a feature vector e. For efficiency, we adopt a faster linear attention mechanism instead of the common tanh-based ones [7]. The linear attention has two stages. **Stage one:** compute the signal matrix $\mathbf{C}_t$ via the following recurrence.

$$\mathbf{C}_t = \begin{cases} h_0 h_0^\top & \text{if } t = 0 \\ (1-\alpha_t)\mathbf{C}_{t-1} + \alpha_t h_t h_t^\top & \text{if}\, t > 0 \end{cases} \tag{8}$$

where $\alpha_t = \sigma(\mathbf{W}_\alpha h_t)$ is the forget gate and $\mathbf{W}_\alpha$ its parameter. **Stage two:** output encoding feature via

$$e = \mathbf{C}_T h_T. \tag{9}$$

The output vector e is the encoded feature vector of s.

4.3 Architecture of Q-Net

Considering that actions with high cumulative rewards shall have close correlations with the current state, we use the inner product of the two object's feature vectors to model the Q-function, that is

$$Q_\theta(s,a) = (e)^\top q_a \tag{10}$$

where $q_a = \mathbf{V}[a]$ is the embedding vector of action a.

4.4 Architecture of Generator

Since Huffman tree uses shorter codes for more frequent items, it results in a faster sampling process and is widely used in NLP tasks [17,18]. To reduce training time, we build a novel neural structure based on Huffman tree. The proposed structure is depicted in Fig. 1(b). The Huffman tree is built according to the popularity of items f which is defined by

$$f_i = \frac{\#\text{ocurr}_i}{\sum_{i=1}^{n} \#\text{ocurr}_i} \tag{11}$$

with $\#\text{ocurr}_i$ being the occurrence number of item i. We assign Huffman code to each node of the tree by the following rules: (a) encode root node by $b_0 = 0$; (b) for a node with code $b_0 b_1 \ldots b_j$, encode its left child by $b_0 b_1 \ldots b_j 0$ and right child by $b_0 b_1 \ldots b_j 1$. Let $z_{b_{0:k}} \in \mathbb{R}^d$ be an embedding vector of a tree node with code $b_{0:k}$. For an item a with code $b_{0:j}$, its generating probability can be computed as follows.

$$g_\theta(a|s) = \prod_{k=0}^{j-1} \underbrace{\left(\sigma(z_{b_{0:k}}^\top e)\right)^{b_{k+1}} \left(1 - \sigma(z_{b_{0:k}}^\top e)\right)^{1-b_{k+1}}}_{\sigma_{n(b_{0:k+1})}} \tag{12}$$

According to above equation, computing $g_\theta(a|s)$ involves calculating $O(j)$ sigmoid functions where j is the length of a's Huffman code. Since the expected length of Huffman codes is $O(\log|A|)$, the time complexity for computing $g_\theta(a|s)$ is $O(d \log|A|)$. Similarly, sampling from $g_\theta(a|s)$ is equivalent to sample $O(j)$ sigmoids which has expected time complexity $O(d \log|A|)$.

Remark 1. The recommendation policy of GCQ can be derived by selecting an optimal action which has highest Q-value among a candidate set, that is

$$\pi(s) = \arg\max_{a \in A} Q_\theta(s,a) \text{ s.t. } A = \{a_i \sim g_\theta(a|s)\}_{i=1}^{c} \tag{13}$$

The recommendation policy can be executed in $O(d \log|A|)$ flops.

4.5 Parameter Inference

Loss Function of the Generator. We use the negative log-likelihood of the generator to evaluate the performance of the generator.

$$nll(\theta) = -\frac{1}{|B|} \sum_{(s,a)\in B} \log g_\theta(a|s). \tag{14}$$

Loss Function of the Q-Net. According to the framework of fitted Q-iteration [1], the loss function of Q-net is the mean square error between the Q-net and its bellman update, namely

$$qloss(\theta) = (Q_\theta(s,a) - r + \gamma \max\{Q_{\theta'}(s',a)|a \in A\})^2 \tag{15}$$

where $A = \{a_i \sim g_{\theta'}(a|s')\}_{i=1}^c$ is the candidate set and $(s,a,s',r) \in B$.

Algorithm 1: Generator Constrained Deep Q-Learning

```
   input : Replay Buffer $B$, size of candidate set $c$, regularizer $\lambda$, number of
           iterations $K$, discount rate $\gamma$, learning rate $\eta$
1  $tree$ = BuildHoffmanTree( $B$ )
2  $\theta_0$ =InitializeParameters( $tree$ )
3  for ( $k = 0; k < K; k++$ ) do
4      $(s, a, s', r)$ = GetRandomSample( $B$ )
5      $A = \{a_i | a_i \sim g_{\theta_k}(a|s'), i \le c\}$
6      $\hat{Q} = r + \gamma \max\{Q_{\theta_k}(s', a_i)|a_i \in A\}$
7      $qloss = \frac{1}{2}\left(\hat{Q} - Q_\theta(s,a)\right)^2$
8      $nll = -\log g_\theta(a|s)$
9      $jointloss = qloss + \lambda nll$
10     $d\theta_k = (Q_{\theta_k}(s,a) - \hat{Q})\nabla Q_{\theta_k}(s,a) - \frac{\lambda}{g_{\theta_k}(a|s)}\nabla g_{\theta_k}(a|s)$
11     $\theta_{k+1} = \theta_k - \eta d\theta_k$
12 end
```

Joint Inference. Since the Q-net and the generator share the same encoder, We jointly train them via iteratively minimizing the following loss.

$$\min_\theta qloss(\theta) + \lambda nll(\theta) \tag{16}$$

where $\lambda > 0$ is a tuning parameter controlling the balance of mean square loss and log-likelihood. The joint loss can be optimized via stochastic gradient descent, as showed in Algorithm 1.

5 Experiments

In this section, we compare the performance of proposed GCQ method with state-of-the-art recommendation methods. We assess the performance of considered methods on both real-world offline datasets and simulated online environments. Besides, empirical studies on the hyperparameter sensitivities and computing time are conducted on several datasets. The baseline methods are listed as follows.

– **MF** [13]: It utilizes the latent factor model to predict the unknown ratings.
– **W&D** [5]: W&D uses wide & deep neural architecture to learn nonlinear latent factors.
– **GRU4Rec** [11]: It applies GRU to model click sequences.
– **DQN**: It recommends items by a deep Q-net. For fairness, We set the Q-net to the same one as the proposed method.
– **DDPG** [8]: DDPG utilizes deterministic policy gradient descent to update parameters.
– **DEERS** [22]: It tries to incorporate a user's negative feedback via sampling from the unclicked items.

5.1 Experiment Settings

We use three publicly available datasets: MovieLens 1M (M1M), MovieLens 10M (M10M) and Amazon 5-core grocery and gourmet food (AMZ) to compare the considered methods. These datasets contain historical ratings of items with scoring timestamps. Now according to timestamps, we can transform the datasets into replay buffers of the form $\{(s_t^u, a_t^u, s_{t+1}^u, r_t^u)\}$.

For simplicity, we set the dimension of user embedding, the dimension of item embedding, and the dimension of hidden states of the proposed neural architectures to the same value d. We call d the model dimension. We set the model dimension $d = 150$, the discount factor $\gamma = 0.9$, the size of sampling size $c = 50$, and the regularizer $\lambda = 0.1$ as default. All these hyperparameters are chosen by cross-validation. The hyperparameters of baseline methods are set to default values.

5.2 Offline Evaluation

According to the temporal order, we use the top 70% tuples in the derived replay buffers for training and hold out the remaining 30% for testing. In an offline environment, we cannot obtain the immediate reward of the recommendation policy. As a result, we cannot use the cumulative reward to evaluate the performance of the compared learning agents. Considering that a Q-net $Q_\theta(s, a)$ with high cumulative reward shall assign large value to clicked items and give small value the ignored ones, $Q_\theta(s, a)$ can be viewed as a scoring function which ranks the clicked items ahead of ignored ones. Thus, we can use the ranking metric such as Recall@k and Precision@k to evaluate the compared methods.

Table 1. Offline Recall@k of compared methods

	M1M			M10M			AMZ		
	Reca@1	Reca@5	Reca@10	Reca@1	Reca@5	Reca@10	Reca@1	Reca@5	Reca@10
DQN	0.0088	0.0416	0.0770	0.0052	0.0248	0.0429	0.0445	0.1877	0.3091
GRU4Rec	0.0079	0.0308	0.0540	0.0054	0.0235	0.0373	0.2735	0.4568	0.543
MF	0.0086	0.0324	0.0561	**0.0074**	0.0262	0.0439	0.2517	0.4359	0.5224
W& D	0.0069	0.0313	0.0519	0.0055	0.0238	0.0389	0.3734	0.5405	0.5982
DEERS	0.0048	0.0257	0.0461	0.0037	0.0193	0.0373	0.2926	0.6013	0.7176
DDPG	0.0083	0.0353	0.0596	0.0045	0.0210	0.0344	0.2359	0.4160	0.4743
GCQ	**0.0110**	**0.0495**	**0.0897**	0.0054	**0.0270**	**0.0539**	**0.3764**	**0.6015**	**0.6747**

Table 2. Offline Precision@k of compared methods

	M1M			M10M			AMZ		
	Prec@1	Prec@5	Prec@10	Prec@1	Prec@5	Prec@10	Prec@1	Prec@5	Prec@10
DQN	0.1543	0.1462	0.1396	0.0734	0.0754	0.0722	0.0523	0.0489	0.0432
GRU4Rec	0.1223	0.1043	0.0910	0.0922	0.0732	0.0588	0.3309	0.1263	0.0798
MF	0.1187	0.0920	0.0807	**0.1207**	0.0907	0.0695	0.3133	0.1220	0.0779
W& D	0.0992	0.0862	0.0740	0.1074	0.0836	0.0641	0.4539	0.1559	0.0920
DEERS	0.0770	0.0789	0.0757	0.0539	0.0582	0.0585	0.3414	0.1680	0.1110
DDPG	0.1598	0.1313	0.1155	0.0727	0.0679	0.0580	0.3016	0.1277	0.0791
GCQ	**0.1789**	**0.1658**	**0.1547**	0.0930	**0.0931**	**0.0930**	**0.4703**	**0.1829**	**0.1092**

To reduce randomness, we run each model five times and report their average performances in Table 1 and Table 2. From the tables, we can see that GCQ consistently outperforms DQN. Since the two methods share the same Q-net, such result shows that GCQ has a lower exploration error during the learning process. GCQ also has higher accuracy than DEERS. The reason is that the proposed encoder is more expressive than DEERS's GRU based one. Compared with DDPG, our GCQ consistently has better accuracy. This is because the policy-gradient-based method DDPG has higher variances during the learning process. Both Table 1 and Table 2 exhibit that GCQ outperforms non-RL methods, namely MF, W&D and GRU4Rec. These results demonstrate that taking the long term reward into consideration can improve the accuracy of recommendation.

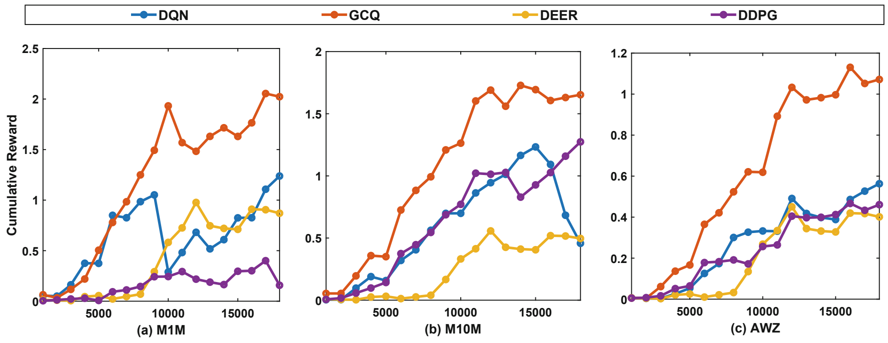


Fig. 2. Cumulative rewards of compared methods v.s. number of iterations

Table 3. Computational time of compared methods

	DQN	DDPG	DEER	GCQ
M1M	116.2 (s)	120.5	129.1	**86.1**
M10M	1175.7	1219.2	1306.3	**870.9**
AWZ	139.9	145.2	155.5	**103.7**

The computational time of compared RL methods is recorded in Table 3. The table exhibits that GCQ takes significantly less computational time in handling the benchmark datasets. This is because GCQ only takes $O(\log|A|)$ flops to make a recommendation decision while the decision complexities of other baseline methods are $O(|A|)$.

5.3 Online Evaluation

To simulate online environment, we train a GRU to model users' sequential decision processes. The GRU takes a user, the user's last clicked 20 items, and a candidate item as input. Then, it outputs the click probability of the candidate item. Such a simulation GRU is widely used in evaluating the online performance of RL-based recommender agent [22]. We split the datasets into the front 10%, the middle 80% and the tail 10% sub-datasets by temporal order. The front sub-dataset is used for initializing the learning agents. The middle sub-dataset is utilized for training the simulation GRU. The simulator will be validated on the tail sub-dataset. After training, we find that the simulator has classification accuracy greater than 75%. Therefore, the simulator quite precisely models a user's click decision. After the simulator is trained, we collect the simulated responses of users and then obtain cumulative reward.

The cumulative reward curves are reported in Fig. 2. From the figure, we find that GCQ yields much higher cumulative rewards than baseline methods. Its superior performance results from the smaller exploration error and better encoder structure. These figures also show that GCQ is more stable than the baseline methods. This confirms that GCQ has a lower exploration error during the learning process.

5.4 Model Stability

We find that the most important hyperparameters include: the model dimension parameter d which controls the model complexity of GCQ; and the size of candidate set c which controls exploration error.

We report Precision@10 of GCQ under different settings of d in Fig. 3(a). Figure 3(b) records Precision@10 of GCQ under different values of c. The experimental results in Fig. 3(a)(b) fluctuate within an acceptable range. This demonstrates the performance of our model is stable.

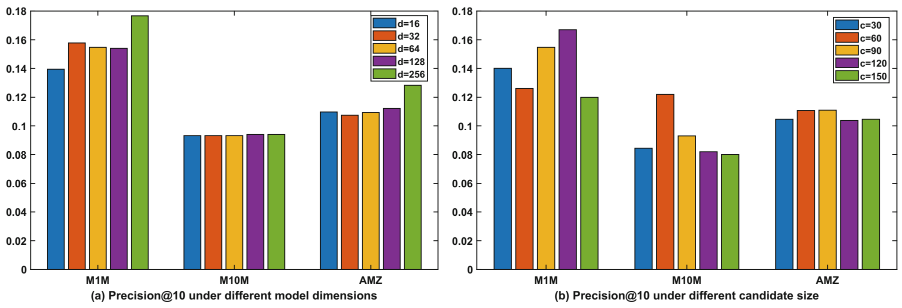


Fig. 3. Precision@10 of GCQ under different settings of hyperparameters

6 Conclusion

We proposed a novel Generator Constrained Q-learning technique for recommendation tasks. GCQ reduce the decision complexity of Q-net from $O(|A|)$ to $O(\log|A|)$. In addition, GCQ enjoys lower exploration error through better characterization of observed data. Further, GCQ employs a new multi-layer encoder to handle long sequences through attention mechanism and skip connection. Empirical studies demonstrate GCQ outperforms state-of-the-art methods both in efficiency and accuracy.

References

1. Antos, A., Szepesvári, C., Munos, R.: Fitted Q-iteration in continuous action-space MDPs. In: NeurIPS (2008)
2. Baird, L.: Residual algorithms: reinforcement learning with function approximation. In: Machine Learning Proceedings 1995, pp. 30–37. Elsevier (1995)
3. Bonadiman, D., Uva, A., Moschitti, A.: Effective shared representations with multitask learning for community question answering. In: ACL (2017)
4. Chen, M., Beutel, A., Covington, P., Jain, S., Belletti, F., Chi, E.: Top-k off-policy correction for a reinforce recommender system. In: WSDM (2019)
5. Cheng, H.-T., Levent Koc, H., et al.: Wide & deep learning for recommender systems. In: Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, pp. 7–10. ACM (2016)
6. Chung, J., Gulcehre, C., Cho, K., Bengio, Y.: Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555 (2014)
7. de Brebisson, A., Vincent, P.: A cheap linear attention mechanism with fast lookups and fixed-size representations. arXiv preprint arXiv:1609.05866 (2016)
8. Dulac-Arnold, G., et al.: Deep reinforcement learning in large discrete action spaces. arXiv preprint arXiv:1512.07679 (2015)
9. Fujimoto, S., Meger, D., Precup, D.: Off-policy deep reinforcement learning without exploration. arXiv preprint arXiv:1812.02900 (2018)
10. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to forget: Continual prediction with LSTM (1999)

11. Hidasi, B., Karatzoglou, A., et al.: Session-based recommendations with recurrent neural networks. arXiv:1511.06939 (2015)
12. Kolter, J.Z.: The fixed points of off-policy TD. In: Advances in Neural Information Processing Systems, pp. 2169–2177 (2011)
13. Koren, Y., Bell, R., Volinsky, C.: Matrix factorization techniques for recommender systems. Computer **42**(8), 30–37 (2009)
14. Kumar, A., Fu, J., et al.: Stabilizing off-policy Q-learning via bootstrapping error reduction. arXiv:1906.00949 (2019)
15. Linden, G., Smith, B., York, J.: Amazon.com recommendations: item-to-item collaborative filtering. IEEE Internet Comput. **1**, 76–80 (2003)
16. Munos, R.: Error bounds for approximate policy iteration. ICML **3**, 560–567 (2003)
17. Peng, H., Li, J., Song, Y., Liu, Y.: Incrementally learning the hierarchical softmax function for neural language models. In: AAAI (2017)
18. Rong, X.: word2vec parameter learning explained. arXiv:1411.2738 (2014)
19. Zaeemzadeh, A., Rahnavard, N., Shah, M.: Norm-preservation: why residual networks can become extremely deep? arXiv preprint arXiv:1805.07477 (2018)
20. Zhang, S., Yao, L., Sun, A., Tay, Y.: Deep learning based recommender system: a survey and new perspectives. CSUR **52**(1), 1–38 (2019)
21. Zhao, X., Zhang, L., Ding, Z., et al.: Deep reinforcement learning for list-wise recommendations. arXiv:1801.00209 (2017)
22. Zhao, X., Zhang, L., Ding, Z., et al.: Recommendations with negative feedback via pairwise deep reinforcement learning. In: SIGKDD (2018)
23. Zheng,G., Zhang, F., Zheng, Z., et al.: DRN: a deep reinforcement learning framework for news recommendation. In: WWW (2018)