THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學
Pao Yue-kong Library
包玉剛圖書館

Copyright Undertaking

This thesis is protected by copyright, with all rights reserved.

By reading and using the thesis, the reader understands and agrees to the following terms:

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the thesis.
2. The reader will use the thesis for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.
3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

IMPORTANT

If you have reasons to believe that any materials in this thesis are deemed not suitable to be distributed in this form, or a copyright owner having difficulty with the material being included in our database, please contact lbsys@polyu.edu.hk providing details. The Library will look into your claim and consider taking remedial action upon receipt of the written requests.

Pao Yue-kong Library, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

http://www.lib.polyu.edu.hk

Personalized Deep Reinforcement Learning based Recommendations

Yu LEI

PhD

The Hong Kong Polytechnic University

2020

The Hong Kong Polytechnic University

Department of Computing

Personalized Deep Reinforcement Learning based Recommendations

Yu LEI

A thesis submitted in partial fulfilment of the requirements

for the degree of Doctor of Philosophy

March 2020

Certificate of Originality

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

________________________(Signed)

_________Lei Yu_________(Name of student)

Abstract

Recommender system is a powerful tool for information filtering, which aims to proactively recommend potentially interesting items to users. The majority of existing recommendation algorithms are essentially *supervised learning* (SL) based approaches, which aim to learn static, passive, and shortsighted predictive models for the single-step recommendation problem. As a result, they are not able to provide satisfactory solutions to the multi-step interactive recommendation problem in more practical scenarios. To address this issue, a promising way is to leverage the *reinforcement learning* (RL) paradigm to build an RL-based recommendation agent. Different from SL-based systems, the RL-based agent aims to learn a dynamic, proactive, and far-sighted recommendation policy that optimizes the cumulative rewards received in the multi-step interactive recommendation process.

In the literature, a number of RL-based recommendation agents, including the early proposed tabular RL agents and the recently proposed deep RL agents, have demonstrated great potential in different recommendation scenarios and datasets. However, there is a fundamental domain-specific problem that has been rarely noticed and investigated in the past. That is *how to effectively model personalization and collaboration in an RL-based recommendation agent*. Personalization means that the agent should model the personalized characteristics of each user as much as possible, while collaboration implies that the agent should model the collaborative relationships (e.g., behavioral similarities) between different users as much as possible.

Both of them are crucial factors to providing high-quality personalized recommendations for the entire user community, and thus to optimizing the overall profit of the recommender system.

The objective of my thesis work is to develop truly personalized RL-based recommendation agents by systematically addressing the issues of personalization and collaboration modeling. Without loss of generality, we adopt value-based deep RL as the basis to conduct our research. We observe that the performance of a value-based deep RL agent is affected by three key components: *the state/action representation module* that transforms raw states/actions to high-level representations, *the action-value prediction module* that outputs action-values based on high-level state/action representations, and *the Q-learning module* that determines how to update the Q-network (i.e., the combination of the first two modules) towards the optimal action-value function corresponding to an optimal policy. Based on these observations, and inspired by the advancements in SL-based recommendation, we develop four novel and effective value-based deep RL recommendation agents. Each of the proposed agents is designed based on substantial improvements in one or more of the three modules by incorporating personalization and collaboration in different ways.

The first work presents User-specific Deep Q-network (UDQN), a two-stage pipeline agent that first constructs latent vector representations of states and actions using matrix factorization (MF) and then estimates action-values based on those representations using Q-learning. In the UDQN agent, the MF-based state/action representation module effectively models personalization and collaboration by mapping all users into a shared latent feature space. The second work describes Graph Convolutional Q-network (GCQN), an end-to-end agent that directly estimates action-values based on the input of graph-structured states and actions. The GCQN agent integrates a graph convolutional network (GCN) based state/action representation module, which successfully models personalization and collaboration by aggregat-

ing valuable features from target user's local neighborhood in the user-item bipartite graph, in terms of feature propagation on the "user-item-user" paths. The third work introduces Social Attentive Deep Q-network (SADQN), which explicitly integrates personalization and collaboration, on the basis of UDQN, by predicting action-values via the combination of a personal action-value function and a social action-value function. The two functions aim to estimate action-values based on the preferences of individual users and of their social friends in the social network, respectively. In particular, the social action-value function successfully models the collaborations between target user and his/her social friends by leveraging social attention to capture the social influence between them. Finally, the fourth work presents Personalized Deep Q-network (PDQN), which is able to estimate fully personalized action-values based on the user-specific information of target user and the general information of state shared by all users. Unlike the UDQN and GCQN agents that implicitly model personalization and collaboration into the state/action representations, PDQN explicitly models them into a brand-new personalized Q-network architecture that consists of a user-specific action-value function and a general action-value function. On the other hand, unlike the SADQN agent, PDQN does not rely on additional social information, which is more domain-free and can be applied to more recommendation scenarios and datasets. Moreover, collaboration is further modeled by PDQN in terms of a novel collaborative Q-learning module.

The extensive and solid experiments on real-world datasets demonstrate that the proposed agents achieve the state-of-the-art performance. More importantly, the ideas, methods, and techniques proposed in this thesis are both insightful and generic, which will promote the advancement of RL-based recommendation, and also bring inspirations to the researchers in other related fields.

Publications Arising from the Thesis

1. **Yu Lei** and Wenjie Li. Interactive Recommendation with User-Specific Deep Reinforcement Learning. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 2019.

2. **Yu Lei**, Zhitao Wang, Wenjie Li, and Hongbin Pei. Social Attentive Deep Q-network for Recommendation. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, 2019.

3. **Yu Lei**, Wenjie Li, Ziyu Lu, and Miao Zhao. Alternating Pointwise-Pairwise Learning for Personalized Item Ranking. In *Proceedings of the 26th ACM International Conference on Information and Knowledge Management (CIKM)*, 2017.

4. **Yu Lei**, Hongbin Pei, Hanqi Yan, and Wenjie Li. Reinforcement Learning based Recommendation with Graph Convolutional Q-network. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, 2020. (Under review)

5. **Yu Lei** and Wenjie Li. Personalized Deep Q-network for Interactive Recommendation. *ACM Transactions on Information Systems (TOIS)*. (Under review)

6. **Yu Lei**, Zhitao Wang, Wenjie Li, Hongbin Pei, and Quanyu Dai. Social Attentive Deep Q-networks for Recommender Systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*. (Received a major revision)

7. Bo Yang, **Yu Lei**, Jiming Liu, and Wenjie Li. Social Collaborative Filtering by Trust. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2017.

8. Zhitao Wang, **Yu Lei**, and Wenjie Li. Neighborhood Interaction Attention Network for Link Prediction. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM)*, 2019.

9. Qiang Chen, Wenjie Li, **Yu Lei**, Xule Liu, and Yanxiang He. Learning to Adapt Credible Knowledge in Cross-Lingual Sentiment Analysis. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2015.

10. Qiang Chen, Wenjie Li, **Yu Lei**, Xule Liu, Chuwei Luo, and Yanxiang He. Cross-Lingual Sentiment Relation Capturing for Cross-Lingual Sentiment Analysis. In *European Conference on Information Retrieval (ECIR)*, 2017.

11. Chengyao Chen, Zhitao Wang, **Yu Lei**, and Wenjie Li. Content-based Influence Modeling for Opinion Behavior Prediction. In *Proceedings of the 26th International Conference on Computational Linguistics (COLING)*, 2016.

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Wenjie Li for her careful guidance, continuous support, and great advice during my Ph.D. study. On the academic level, Prof. Li taught me a lot in how to become a qualified researcher in the computer science area. Under her supervision, I learned how to define a research problem, find a solution to it, and finally present the results to other researchers. On a personal level, Prof. Li inspired me deeply by her hardworking and passionate attitude. It is my honor to be her student, and I could not imagine how I can survive the Ph.D. study without her care and help.

I would like to thank my kind and talented lab mates that include Zhitao Wang, Chengyao Chen, Yanran Li, Ziqiang Cao, and so forth for their continued support. I am lucky to join Prof. Li's group, and this dissertation would not have been possible without their constructive suggestions and help. I would also like to thank Hongbin Pei, Qiang Chen, Hanqi Yan, Ziyu Lu, Miao Zhao, and Quanyu Dai for their helpful suggestions and discussions.

Last but not the least, I dedicate this dissertation to my family for all their constant and unconditional love. I would like to express my deepest gratitude to my parents and sisters, who raised me with all their love and care, and gave me a lot of freedom to pursue my own interests. I would also like to thank my loving girlfriend Ms. Minghui Zhang, who gave me faithful support during this hard journey.

Table of Contents

Certificate of Originality **iii**

Abstract **v**

Acknowledgements **xi**

List of Figures **xix**

List of Tables **xxi**

1 Introduction **1**

1.1 Research Background 1

1.2 Research Motivation 4

1.3 Research Overview and Contributions 9

1.4 Structure of Thesis 16

2 Literature Review **17**

2.1 Collaborative Filtering 17

2.1.1 Matrix Factorization 18

2.1.2 Ranking-oriented Collaborative Filtering 19

2.1.3 Neural Collaborative Filtering 20

2.1.4 Social Collaborative Filtering 21

2.1.5 Summary 23

2.2 Reinforcement Learning 24

2.2.1 Tabular Reinforcement Learning 24

2.2.2 Deep Reinforcement Learning 25
2.2.3 Summary 27

3 User-specific Deep Q-network: Modeling Personalization and Collaboration via MF-based Representation Module 29

3.1 Introduction 30
3.2 Preliminaries 32
3.2.1 Problem Definition 32
3.2.2 Q-learning 32
3.3 User-specific Deep Q-network 34
3.3.1 The Markov Decision Process (MDP) 34
3.3.2 The Multi-MDP Reinforcement Learning Task 35
3.3.3 User-specific Latent States based on Matrix Factorization . . . 36
3.3.4 The UDQN Model 38
3.4 Experiments on Explicit Feedback Recommendation Tasks 42
3.4.1 Experimental Settings 42
3.4.2 Performance Comparison 45
3.4.3 Parameter Analysis 49
3.4.4 The Results of Cross-validation 51
3.5 Experiments on Implicit Feedback Recommendation Tasks 52
3.5.1 Experimental Setup 53
3.5.2 Performance Comparison 56
3.5.3 Parameter Analysis 58
3.6 Conclusions and Discussions 60

4 Graph Convolutional Q-network: Modeling Personalization and Collaboration via GCN-based Representation Module 63

4.1 Introduction 63

4.2 Preliminaries . . . 67
4.3 Graph Convolutional Q-network . . . 68
4.3.1 Representing States and Actions as Graphs . . . 68
4.3.2 The GCQN Model . . . 69
4.3.3 Training Algorithm . . . 73
4.4 Experiments . . . 74
4.4.1 Experimental Setup . . . 75
4.4.2 Comparison Results . . . 78
4.4.3 Model Analysis . . . 79
4.5 Conclusions and Discussions . . . 82
5 Social Attentive Deep Q-network: Improving Personalization and Collaboration via Social Attention 85
5.1 Introduction . . . 86
5.2 Preliminaries . . . 88
5.2.1 Problem Formulation . . . 88
5.2.2 Reinforcement Learning . . . 89
5.2.3 Deep Q-network . . . 90
5.3 Social Attentive Deep Q-networks . . . 91
5.3.1 SADQN: A Linear Fusion Model . . . 92
5.3.2 SADQN++: A Deep Fusion Model . . . 95
5.3.3 Training Algorithm . . . 97
5.4 Experiments . . . 99
5.4.1 Experimental Setup . . . 99
5.4.2 Performance Comparison against Baselines . . . 104
5.4.3 The Impact of Social Influence . . . 106
5.4.4 Model Analysis . . . 108

5.5 Conclusions and Discussions 112

6 Personalized Deep Q-network: Integrating Personalized Network Architecture with Collaborative Learning Objective 113

6.1 Introduction 114

6.2 Preliminaries 116

6.2.1 Interactive Recommendation 116

6.2.2 Markov Decision Process 117

6.2.3 Deep Q-network 119

6.3 Personalized Deep Q-network 120

6.3.1 Motivation and Idea 120

6.3.2 The PDQN Model 123

6.3.3 Personalized Q-learning 128

6.3.4 Personalized Q-learning with Collaborative Regularizer 129

6.4 Experiments 132

6.4.1 Experimental Setup 133

6.4.2 Comparison Results 136

6.4.3 Parameter Analysis 138

6.5 Conclusions and Discussions 139

7 Conclusions and Suggestions for Future Research 141

7.1 Summary of Contributions 143

7.1.1 UDQN: Modeling Personalization and Collaboration via MF-based Representation Module 143

7.1.2 GCQN: Modeling Personalization and Collaboration via GCN-based Representation Module 144

7.1.3 SADQN: Improving Personalization and Collaboration via Social Attention 145

7.1.4 PDQN: Integrating Personalized Network Architecture with Collaborative Learning Objective 146

7.2 Future Work 146

Bibliography **149**

List of Figures

1.1 Modeling the agent-user interaction in recommendation as the agent-environment interaction in RL. 4

1.2 The Multi-MDP RL task in real-world recommendation scenarios. Each MDP_u is corresponding to the interactive recommendation process of user u. 5

1.3 Illustration of a typical value-based deep RL agent. The agent consists of three key components: *the state/action representation module* that transforms raw states/actions to high-level representations, *the action-value prediction module* that outputs action-values based on high-level state/action representations, and *the Q-learning module* that determines how to update the Q-network (i.e., the first two modules) towards the optimal action-value function. 7

1.4 Illustration of the main contributions of this thesis, which shows how we model personalization and collaboration into the agent in each of our four works. 10

3.1 The basic framework of UDQN. 40

3.2 Effect of the discount factor γ on the performance of UDQN. 46

3.3 Effect of the experience replay on the performance of UDQN. 50

3.4 Effect of the parameter T_{train} on the performance of UDQN. 50

3.5 Effect of the discount factor γ on the performance of UDQN in terms of cross-validation. 52

3.6 Performance of UDQN-Concat for different k. 59

3.7 Performance of UDQN-Concat for different T. 60

4.1 A toy example to illustrate how to build a graph-structured state for target user u. (a) The whole user-item bipartite graph G. (b) The sub-graph $G(i_1)$ that consists of item i_1 and its neighborhood in G. (c) Building a graph-structured state $s = \{G(i_1), ..., G(i_4)\}$ based on the raw state $s = \{i_1, ..., i_4\}$. 66

4.2 Overview of the GCQN model. 70

4.3 Comparison for different $T \in \{5, 10, 15, 20\}$. Our GCQN consistently shows remarkable improvements over AttGRU-Q. 80

4.4 Comparison of running time of reinforcement learning based methods. The training cost of GCQN is close to DEERS, nearly 1.5 times higher than AttLSTM and AttGRU, and 2 times higher than LSTM and GRU. This indicates that the significant performance gain of GCQN is achieved with acceptable cost. 81

5.1 (a) The user-agent interaction in RL based recommender systems. (b) The basic architecture of DQN. 89

5.2 SADQN: a linear fusion model. 93

5.3 SADQN++: a deep fusion model. 96

5.4 Performance comparison against baselines on both **cold-start** and **warm-start** recommendations. The mean (bar) and standard deviation (line) of HR and NDCG@10 metrics on all three datasets are shown. The proposed SADQN++ model shows the best performance in all cases. 105

5.5 Comparison of the run-time of RL-based methods. 110

5.6 Effect of the discount factor γ on the performance of SADQN^P. . . . 111

5.7 Effect of the experience replay on the performance of SADQN^P. . . . 112

6.1 The basic DQN agent. 119

6.2 The three learning paradigms for the multi-MDP task: the existing RL and MARL, and our proposed PRL. 123

6.3 The proposed PDQN agent. 124

6.4 The impact of the CR regularization parameter λ and the size of collaborative neighborhood N on the performance of PDQN-cr on each dataset. 138

List of Tables

3.1 The statistics of datasets 43
3.2 Performance comparison on Task I. 47
3.3 Performance comparison on Task II. 48
3.4 The results of cross-validation on ML100K dataset. 52
3.5 The statistics of datasets 53
3.6 Performance comparison on implicit-feedback top-k recommendation task. 57
4.1 The statistics of datasets. 75
4.2 The average reward received in T-step episodes ($T = 20$). 78
4.3 The results of GCQN with different aggregators. 81
4.4 The results of GCQN with different sample size L. 82
5.1 The Statistics of Datasets 99
5.2 The Experimental Results of Different Variants of SADQN 107
5.3 The Results of SADQN with Different Attention Mechanisms 109
6.1 The statistics of the Amazon datasets. 133
6.2 The parameter settings of PDQN-cr. 136
6.3 Performance comparison of all methods, in terms of the AvgReward. The best performing method is highlighted in bold font (always PDQN-cr). The best performing baseline is marked with "*" (always DQN). 137

Chapter 1

Introduction

1.1 Research Background

Recommender systems refer to a class of intelligent systems that aim to proactively provide users with potentially interesting items (information, services, or products). Nowadays, recommender systems have been successfully applied to almost all of modern websites, such as the product recommender system on Amazon, the video recommender system on YouTube, and the friend recommender system on Facebook. These recommender systems are able to not only increase the business value significantly from the viewpoint of websites, but also improve the user experience fundamentally from the viewpoint of users. Thus, the research on recommender systems is of significant social value and economic value.

The core of a recommender system is the recommendation algorithm/method it adopted, which determines what items should be recommended to users. Over the past two decades, a large number of studies on recommendation algorithms have been conducted by researchers in both academy and industry, across different research areas including artificial intelligence, machine learning, data mining, information retrieval, and so forth. A variety of recommendation algorithms have been proposed in the literature and successfully applied to real-world recommender systems.

Among the existing recommendation algorithms, *collaborative filtering* (CF) is

one of the most prominent and popular approaches to recommendation. CF methods leverage the observed user-item feedback data to infer users' preferences, in terms of similarities of users or items, and thus make personalized recommendations for diverse users [36, 28, 104, 140]. In particular, *matrix factorization* (MF) models, one class of model-based CF methods that map users and items into a shared latent feature space and predict user-item relevance scores using inner product of feature vectors of users and items, have dominated the rating prediction and top-k recommendation tasks in various domains (e.g., movies, music) for a few years [102, 57, 59, 43]. Unfortunately, CF methods heavily suffer from the issues of cold-start and data sparsity that are pervasive in real-world recommender systems. To address such issues, many researchers propose to leverage the available user-user social networks to help infer users' preferences and develop social CF methods [82, 80, 53, 83, 107, 152, 74, 32, 17], while some other researchers suggest to exploit the item-item knowledge graphs to improve user preference modeling and design knowledge-graph-based CF methods [139, 137, 50, 118, 136, 138, 154, 155]. On the other hand, the majority of CF methods are intrinsically linear models or heuristic methods that cannot effectively reveal the complicated user behavior patterns behind the observed user-item feedback data. To alleviate this issue, a large number of neural CF methods have been proposed in recent years, which leverage neural networks to capture the complex user-item interaction patterns and learn meaningful representations of users and items to help make recommendations [156, 21, 133, 37, 42, 114, 142].

Despite their successes, the aforementioned recommendation methods are intrinsically non-interactive and shortsighted. Most of them follow the same *supervised learning* (SL) paradigm that aims to learn static, passive, and shortsighted predictive models from user historical data and make predictions for the single-step recommendation problem only [1, 59, 58, 100]. In more practical recommendation scenarios, however, the recommender usually interacts with the target user for multiple steps,

and accordingly makes a sequence of interactive recommendations. Importantly, a key feature of such interactive recommendation process is that the recommendations provided currently may have huge impacts on the quality of future recommendations. More specifically, the item recommended at current step may not be liked by the user, but the received feedback might provide valuable information about his/her preferences, which can help the recommender make better recommendations at future steps [101]. Besides, the item recommended at current step may change the user's preferences dynamically, which also affects the decisions and results of future recommendations. For such multi-step interactive recommendation scenarios, the existing methods fail to consider the important influences from earlier steps to later steps, which can merely generate greedy and shortsighted recommendations that fit the estimated short-term preferences of the user at each single step. Instead, a more desirable method should effectively model the user's long-term preferences towards future and provide farsighted recommendations that optimize the cumulative rewards received in the multi-step interactive recommendation process.

A potential approach to interactive recommendation is to leverage the *reinforcement learning* (RL) paradigm [119] to build an RL-based recommendation agent. Compared to SL-based recommendation methods, a notable advantage of RL-based agents is that they are able to not only discover the user's dynamic preferences via continuous user-agent interactions, but also learn proactive and farsighted recommendation policies in a trial-and-error fashion. The objective of this thesis is to develop effective RL-based recommendation agents that can provide high-quality interactive recommendations to users. The following research questions will be answered in this thesis. *What are the fundamental domain-specific challenges of applying RL to recommendation? How to address the challenges and develop effective RL-based recommendation agents? Do the developed RL-based recommendation agents achieve better performance than the state-of-the-art recommendation methods?*

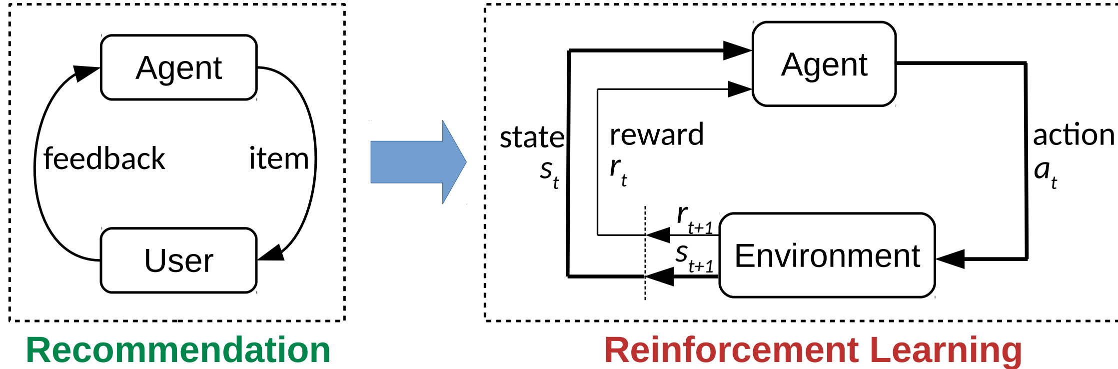


Figure 1.1: Modeling the agent-user interaction in recommendation as the agent-environment interaction in RL.

1.2 Research Motivation

With the RL paradigm [119], the agent-user interaction in recommendation can be naturally modeled as the agent-environment interaction in RL, as shown in Figure 1.1. This gives rise to an episodic RL task, where in each episode, the agent interacts with the environment (corresponding to the interactive recommendation process of target user u) at discrete time steps $t = 0, ..., T-1$. At each time step t, the agent observes a state s_t (describing the dynamic preferences of user u at time step t) of the environment, then takes an action a_t (i.e., recommends an item) according to its policy π (indicating how to choose actions given states). One time step later, as a result of its action, the agent receives a numerical reward r_{t+1} (a signal that reflects the feedback on item a_t given by user u) and a new state s_{t+1} from the environment. More formally, the environment can be mathematically described by a *Markov decision process* (MDP), that is, a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is the state-transition function, and $\mathcal{R} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function. The MDP formulation provides well guarantees for applying standard RL algorithms to solve the RL task, that is, to estimate an optimal policy π^* that maximizes the cumulative reward received during the entire interactive recommendation process.

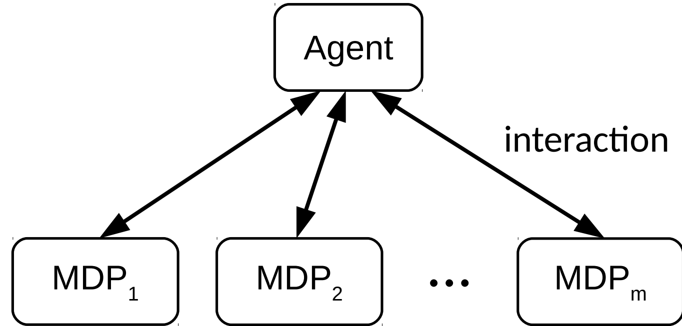


Figure 1.2: The Multi-MDP RL task in real-world recommendation scenarios. Each MDP_u is corresponding to the interactive recommendation process of user u.

Unfortunately, this is far from the desirable solution that can really work to make meaningful interactive recommendations in practice. A real-world recommender system usually involves multiple users, where each user u's interactive recommendation process is actually a unique MDP, denoted by MDP_u. This gives rise to a non-typical multi-MDP RL task, as illustrated in Figure 1.2, where the agent needs to interact with the multiple MDPs of all users. Since the involved users usually have diverse preferences and behaviors over time steps, their MDPs may vary remarkably in terms of both state-transition functions and reward functions. In other words, similar users' MDPs may have relatively close dynamics, but dissimilar users' MDPs may have distinctly different dynamics. As a result, compared to typical single-MDP task, it is much more difficult for the agent to solve the multi-MDP task, that is, to learn an effective policy that makes personalized recommendations for all involved users. To be more specific, *there are two fundamental domain-specific challenges in designing a personalized RL agent for the multi-MDP task in recommendation:*

- **Personalization Modeling.** *Personalization*[1] is probably the most important factor of any real-world recommender system, which is crucial to attracting

[1]In this thesis, when we discuss personalization, we always refer to the concept that a recommender should make personalized recommendations to different users according to their different preferences. Currently, we only leverage users' specific feedback data to model users' preferences. In future work, we will explore to model users' preferences by leveraging the additional user profile. The performance of the proposed methods is expected to be further improved.

user attention and improving user experience. It means that the agent should model the specific personalized characteristics of each user as much as possible, in order to produce personalized user-specific recommendations to each user. In other words, the agent is required to reveal the possible differences among the multiple MDPs of all users.

- **Collaboration Modeling.** *Collaboration* is another crucial factor to building a personalized recommender system in practice, since it offers important clues to the agent to make novel recommendations for each target user, by discovering his/her unknown interests from similar neighbors (i.e., collaborators). It implies that the agent should model the underlying collaborative relationships (e.g., behavioral similarities) between different but similar users, in order to make more effective personalized recommendations for the entire user community. In other words, the agent is required to discover the possible similarities among the multiple MDPs of all users.

The above challenges are aggravated by the pervasive issues of user cold-start[2] and data sparsity[3] in real-world recommender systems. How to effectively address these challenges, that is, how to properly model personalization and collaboration, is vital to building a successful personalized RL-based recommendation agent.

However, we find that the prior works in RL-based recommendation have paid little attention to the problems of personalization and collaboration modeling. Most of the existing RL-based recommendation agents [106, 120, 160, 158, 19, 20, 163] follow the same approach that treats all involved users as a single virtual user and directly learns a unified policy $\pi(s)$ based on the transition data collected uniformly from all MDPs , which outputs actions dependent only on the state s (usually, a

[2]That is to say, the new user who has no feedback data.

[3]That is to say, most users only have feedback on a very small fraction of items, leading to the extreme sparsity of observed user-item feedback data.

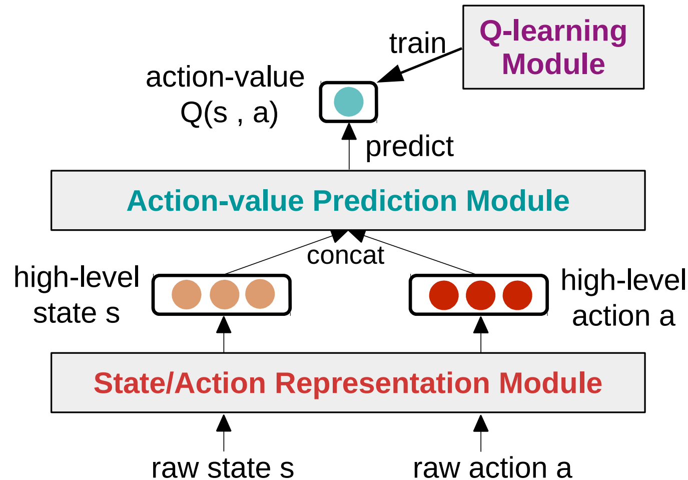


Figure 1.3: Illustration of a typical value-based deep RL agent. The agent consists of three key components: *the state/action representation module* that transforms raw states/actions to high-level representations, *the action-value prediction module* that outputs action-values based on high-level state/action representations, and *the Q-learning module* that determines how to update the Q-network (i.e., the first two modules) towards the optimal action-value function.

sequence of user-consumed items), without any explicit consideration of user-specific information. Although this common adopted approach is straightforward and can be easily implemented in practice, it has an intrinsic weakness that fails to model the key factors of personalization and collaboration. More specifically, this approach is not able to capture the diverse long-term preferences of different users if they are in the same current state s (usually appears at earlier time steps in the interactive recommendation process), since the unified policy $\pi(s)$ will make the same action in state s. On the other hand, it simply treats all the other users equally as the collaborators of each target user, which is not able to effectively discover similar preferences from his/her truly collaborators. A few recently proposed works [76, 75] make an extension to the basic approach by incorporating the additional user id into the state s. However, such simple way of exploiting the user-specific information only leads to very limited and coarse-grained modeling of personalization and collaboration.

In this thesis, we attempt to model personalization and collaboration into RL-

based recommendation agents in a more systematical and fine-grained way. We start our research with some observations on what key components affect the performance of an RL agent. Without loss of generality, we use value-based deep RL as the basic framework to conduct our research, as it is more general and applicable than the policy-based deep RL[4]. A typical value-based deep RL agent utilizes a deep neural network, referred to as Q-network, to approximate the optimal action-value function Q^* (corresponding to an optimal policy π^*) [119, 90] via Q-learning based algorithms [145]. As show in Figure 1.3, it consists of three key components: *the state/action representation module* that transforms raw states/actions to high-level representations, *the action-value prediction module* that outputs action-values based on high-level state/action representations, and *the Q-learning module* that determines how to update the Q-network (i.e., the combination of the first two modules) towards the optimal action-value function.

More specifically, as pointed out in [119], the performance of any RL system is highly dependent on the state/action representations adopted in the system. Generally speaking, good representations should capture sufficient and useful task-related information, in order to facilitate the agent to complete the specific decision making task of interest. In fact, the impressive performance of deep RL agents in many game-based tasks, such as Atari [90, 144, 124] and Go [110, 113], are largely because that they successfully learn useful high-level state representations from raw image inputs by using deep Convolutional Neural Networks (CNNs). On the other hand, the network architecture of the action-value prediction module is also important to the estimation of action-values. For instance, the Dueling Deep Q-network (DDQN) [144], which integrates both value function and advantage function to estimate action-values, has shown better performance than the original Deep Q-network (DQN) [90].

[4]In spite of this, the ideas, concepts and techniques proposed for value-based agents can be readily transferred to policy-based agents with reasonable modifications.

Furthermore, the Q-learning module also plays an critical role in the policy learning process. A number of improvements have been made from the perspective of learning algorithm, such as Double DQN [124] and Robust DQN [19].

These observations motivate us to raise an enlightening question: *Can we effectively model personalization and collaboration into the state/action representation module, the action-value prediction module, or the Q-learning module, to improve the performance of a value-based deep RL recommendation agent?*

1.3 Research Overview and Contributions

To answer the above question, in this thesis we develop four novel and effective value-based deep RL recommendation agents. Each of them is designed based on substantial improvements in one or more of the aforementioned three modules by incorporating personalization and collaboration in different ways. For each proposed agent, we use one work to describe its technical details and verify its efficacy.

In work 1, we present a User-specific Deep Q-network (UDQN) agent [64, 63], which first constructs latent vector representations of states and actions using matrix factorization (MF) and then estimates action-values based on those representations using Q-learning. In work 2, we describe Graph Convolutional Q-network (GCQN) [66], an end-to-end agent that directly estimates action-values based on the input of graph-structured representations of states and actions by effectively processing them using a variant of graph convolutional network (GCN). In the first two works, we successfully model personalization and collaboration into the state/action representation module by leveraging the techniques of MF and GCN, respectively. In work 3, we introduce a Social Attentive Deep Q-network (SADQN) agent [67, 68], which is able to estimate action-values based on both personal preferences and social neighbors' preferences by using personal and social action-value functions, respectively. In

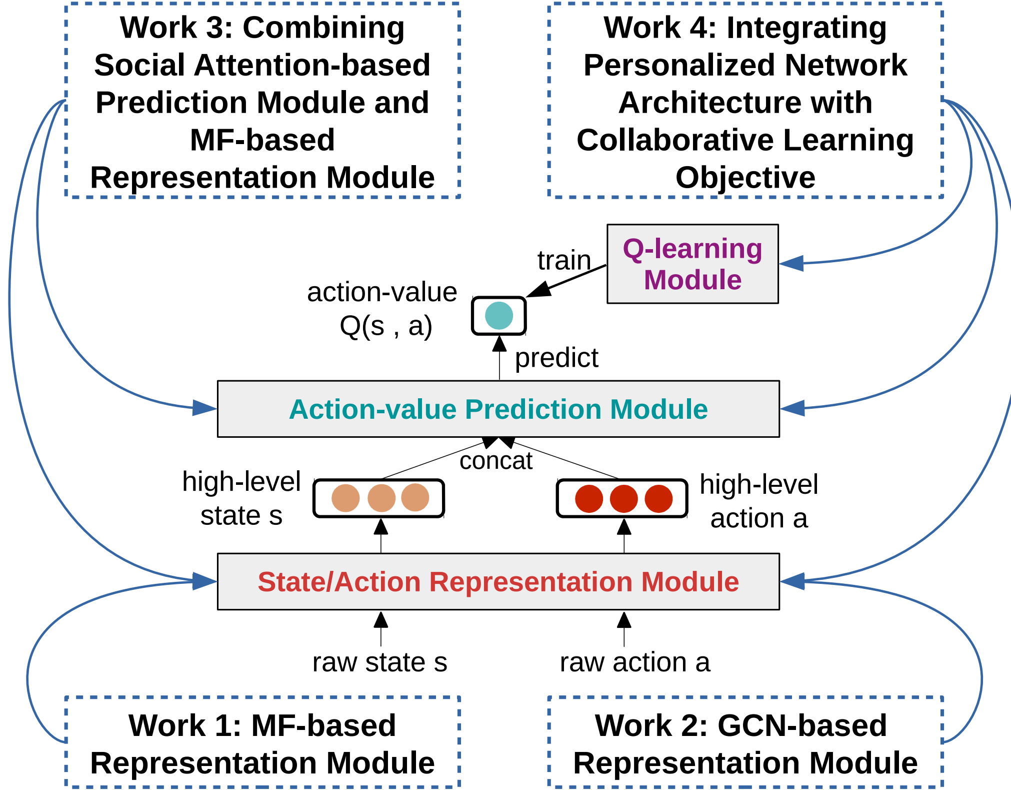


Figure 1.4: Illustration of the main contributions of this thesis, which shows how we model personalization and collaboration into the agent in each of our four works.

this work, on the basis of UDQN, we further model personalization and collaboration into the action-value prediction module by successfully leveraging social attention to capture the social influence between target user and his/her social neighbors in the social network. In work 4, we present Personalized Deep Q-network (PDQN) [62], an end-to-end agent that estimates action-values based on both the user-specific information of user and the general information of state by using user-specific and general action-value functions, respectively. Unlike the above three agents, PDQN utilizes a brand-new personalized Q-network architecture to explicitly model personalization and collaboration, which is able to learn a fully personalized recommendation policy. Moreover, collaboration is further modeled by PDQN in a collaborative Q-learning objective.

Figure 1.4 illustrates briefly how we model personalization and collaboration into the agent in each of the four works. Each work is elaborated in one chapter of this thesis (from Chapter 3 to Chapter 6). Next, we provide a brief introduction to each work and summarizes our main contributions.

Work 1: UDQN: Modeling Personalization and Collaboration via MF-based Representation Module [64, 63]

Existing RL-based recommendation agents are significantly limited by a common weakness that the key factors of personalization and collaboration are not effectively modeled. In this work, we propose a novel way to model the factors of personalization and collaboration into deep RL agents. We first construct user-specific latent vector representations of states and actions by employing matrix factorization (MF), a well adopted collaborative filtering (CF) technique in traditional recommendation problems. After that, we propose User-specific Deep Q-network (UDQN) to approximate the optimal action-value function (corresponding to an optimal recommendation policy) based on the constructed latent representations using Q-learning. Finally, we validate the effectiveness of our approach with comprehensive experimental results and analysis on both explicit-feedback and implicit-feedback recommendation tasks.

Contributions:

- This work is the first attempt to explore the combination of CF and RL for recommendation problems. We seamlessly integrates the ideas of CF and RL by designing an MF-based state/action representation module for the RL-based recommendation agent.
- The proposed MF-based representation module successfully models the factors of personalization and collaboration by mapping all users into a shared latent feature space, and produces informative MF-based vector representations of

states and actions for policy learning.

- We develop a two-stage pipeline agent, named UDQN, which is able to learn an effective recommendation policy based on the constructed MF-based state/action representations through Q-learning.

- We verify the efficacy and capability of UDQN with comprehensive experimental results and analysis on real-world datasets, in terms of both explicit-feedback and implicit-feedback recommendation tasks.

Work 2: GCQN: Modeling Personalization and Collaboration via GCN-based Representation Module [66]

In this work, we propose an alternative way to model personalization and collaboration by building graph-structured representations of states and actions according the user-item bipartite graph. We develop an effective end-to-end agent, termed Graph Convolutional Q-network (GCQN), which is able to directly approximate the optimal action-value function based on the input of graph-structured representations. In particular, GCQN successfully leverages a variant of graph convolutional network (GCN) to transform the low-level graph-structured representations to higher-level vector representations of states and actions. We show that GCQN achieves significant improvements over the existing methods, across different datasets and task settings, with acceptable computation cost.

Contributions:

- This is the first work that incorporates GCN to design an RL-based recommendation agent. We propose a GCN-based state/action representation module that is able to transform the graph-structured representations to meaningful vector representations.

- The proposed GCN-based representation module successfully models the factors of personalization and collaboration by aggregating valuable features from target user's local neighborhood in the user-item bipartite graph, in terms of the feature propagation on the "user-item-user" paths.
- We develop an end-to-end agent, termed GCQN, which is able to directly approximate the optimal action-value function (corresponding an optimal recommendation policy) based on the input of graph-structured representations.
- We conduct extensive experiments on three real-world datasets. The results demonstrate that: (1) the proposed GCN-based representation module helps a lot in learning farsighted recommendation policies, and (2) GCQN is effective, robust and efficient, which achieves significant performance gains over state-of-the-art baselines with reasonable computation cost.

Work 3: SADQN: Improving Personalization and Collaboration via Social Attention [67, 68]

In this work, we propose an effective way to address the issues of data sparsity and cold-start of existing RL-based recommendation agents, by leveraging the available social network among users to promote policy learning. Specifically, we develop a Social Attentive Deep Q-network (SADQN) agent to learn recommendation policies based on the preferences of both individual users and their social neighbors, by successfully utilizing social attention to model the social influence between them. SADQN further models the factors of personalization and collaboration, on the basis of UDQN, by utilizing a combined action-value prediction module that consists of a personal action-value function and a social action-value function. Moreover, we propose an enhanced variant of SADQN, termed SADQN++, to model the complicated and diverse trade-offs between personal preferences and social influence for

all involved users, making the agent more powerful and flexible in learning optimal policies. The solid experimental results on real-world datasets demonstrate that the proposed SADQNs remarkably outperform the state-of-the-art agents, with reasonable computation cost.

Contributions:

- We make the first attempt to improve the performance of RL-based recommendation agents by effectively utilizing available social networks of users.
- We develop a two-stage pipeline agent, termed SADQN, which further models the factors of personalization and collaboration, on the basis of UDQN, by estimating action-values based on the combination of a personal action-value function and a social action-value function. The personal and social action-value functions aim to estimate action-values based on the preferences of individual users and social friends, respectively.
- In particular, the social action-value function successfully models the collaborations between target user and his/her social neighbors by leveraging social attention to capture the social influence between them.
- Further, we develop an enhanced variant of SADQN, termed SADQN++, to model the complicated and diverse trade-offs between personal preferences and social influence for all involved users, making the agent more powerful and flexible in learning optimal policies.
- The solid experimental results on real-world datasets demonstrate that the proposed SADQNs remarkably outperform the state-of-the-art agents, especially in the cold-start recommendation scenario.

Work 4: PDQN: Integrating Personalized Network Architecture with Collaborative Learning Objective [62]

In this work, we provide a more explicit and effective way to model personalization and collaboration in RL-based recommendation agents. We develop an end-to-end agent, term Personalized Deep Q-network (PDQN), which is able to learn a fully personalized policy that makes recommendations based on both the user-specific information of a particular user and the general information of state shared by all users. More specifically, PDQN estimates personalized action-values based on the linear combination of two action-value functions: the user-specific action-value function and the general action-value function that emphasize on modeling personalization and collaboration, respectively. Furthermore, we propose a novel collaborative Q-learning objective to further model the collaborations between similar neighbors, which is able to make the PDQN agent more effective in approximating the optimal personalized policy. We show that PDQN achieves significant performance gains over the existing methods on several real-world datasets.

Contributions:

- We propose a novel end-to-end agent, named PDQN, which is able to learn a fully personalized recommendation policy that depends on both the user-specific information of individual user and the general information of state shared by all users.

- Unlike the UDQN and GCQN agents that model personalization and collaboration implicitly into state/action representations, PDQN explicitly models them into the whole Q-network architecture, by estimating personalized action-values based on the combination of a user-specific action-value function and a general action-value function. Besides, unlike the SADQN agent, PDQN does not rely on additional social information, which can be applied to more

recommendation scenarios and datasets.

- We design a novel collaborative Q-learning objective to further model collaboration between similar neighbors, which is able to make PDQN to approximate the optimal personalized policy more effectively.

- We conduct solid experiments on several real-world datasets to validate our methods. The results sufficiently verify the efficacy of both the personalized Q-network architecture and the collaborative Q-learning objective.

1.4 Structure of Thesis

The rest of this thesis is structured as follows. In Chapter 2, we provide a comprehensive literature review, which surveys the existing studies that are closely relevant to the research in this thesis. We also point out the connections and differences between the existing works and ours. From Chapter 3 to Chapter 6, we orderly introduce the aforementioned four RL agents, and validate them on real-world datasets with solid experimental results and analysis. Finally, Chapter 7 summarizes the proposed methods, findings, conclusions, and contributions of this thesis. The potential future extensions of current studies are suggested at last.

Chapter 2

Literature Review

In this chapter, we survey the existing studies that are closely relevant to the research in this thesis. We categorize the prior works according to the basic types of their proposed recommendation methods, including (generalized) collaborative filtering and reinforcement learning.

2.1 Collaborative Filtering

Collaborative filtering (CF) is probably the most prevalent and successful approach to building personalized recommender systems. The core idea behind CF is to infer users' personal preferences from their nearest neighbors (i.e., collaborators), and thus make personalized recommendations for all users. As CF relies only on the observed user-item feedback data (rating, watching, clicking, etc.), it does not limited to specific recommendation domains, unlike content-based methods [1].

Over the past two decades, a large number of CF-based methods have been proposed in the recommendation literature. Early CF methods are essentially some memory-based (or heuristic-based) algorithms, which aim to predict the rating of a given user-item pair based on the aggregation of the ratings of other similar users (user-based CF), similar items (item-based CF), or both of them (Hybrid CF) [36, 99, 10, 104, 28, 140, 54]. These memory-based CF methods have to compute the

similarities of users or items over the entire user-item matrix, which usually cannot be applied to very large datasets. Moreover, their prediction accuracy is significantly reduced by cold-start and data sparsity, since the computed similarities on very sparse feedback data cannot reflect the real relationships between users or items. To improve the prediction accuracy and scalability, researchers incorporate the techniques in machine learning such as matrix factorization, learning to rank and neural networks, and design a large number of model-based CF methods that aim to learn an accurate prediction model from data. On the other hand, to alleviate the issues of cold-start and data sparsity, many researchers exploit additional data sources of the available social networks to help infer users' preferences and develop a number of social CF methods. We will discuss these methods in the following.

2.1.1 Matrix Factorization

Matrix factorization (MF) is a model-based CF technique, which maps all users and items into a shared latent feature space and makes rating predictions based on the inner product between the feature vectors of users and items. These feature vectors are trained over the observed user-item feedback data by minimizing certain squared loss functions. Over the past decade, MF-based models have dominated the rating prediction and top-k recommendation tasks in academy, and have been successfully applied to real-world recommender systems in industry.

MF for Rating Prediction. A lot of MF models have been proposed for the rating prediction task in explicit feedback datasets [116, 98, 102, 103, 162, 57, 59]. A representative MF model is the probabilistic matrix factorization (PMF) proposed by Salakhutdinov and Mnih [102], which alleviates the over-fitting problem by defining Gaussian priors on the latent feature vectors. The same authors propose Bayesian PMF [103] to further avoid over-fitting, which can be efficiently trained with Markov chain Monte Carlo methods and achieve higher prediction accuracy. Koren [57]

propose another famous MF model, SVD++, which achieves dominant performance in the rating prediction task and win the 2008 Netflix Prize [1]. The core prediction model of SVD++ consists of: the inner product of the user and item feature vectors, the average rating in the observed data, the user bias parameter, the item bias parameter, and the implicit feedback term.

MF for Top-k Recommendation. On the other hand, a number of MF models have been proposed for the top-k item recommendation task in implicit feedback datasets [49, 94, 43]. Hu et al. [49] investigate the special properties of implicit feedback datasets in a TV show recommender system, and propose a MF model by incorporating a confidence factor of users' positive interests into the squared loss function. Pan et al. [94] improve the performance of MF for implicit feedback recommendation by proposing several negative example weighting/sampling strategies, including uniform weighting/sampling, user-oriented weighting/sampling, and item-oriented weighting/sampling. He et al. [43] propose a fast learning algorithm based on element-wise alternating least squares (eALS) technique for online MF in implicit feedback recommendation.

2.1.2 Ranking-oriented Collaborative Filtering

The recommendation tasks can be essentially formulated as a personalized item ranking problem. Many researchers integrate the techniques of both CF and learning-to-rank [77], and design a number of ranking-oriented CF (or called collaborative ranking) methods for the item ranking problem. From the perspective of ranking, the aforementioned MF models can be regarded as pointwise ranking-oriented CF methods, since they employ the pointwise squared error loss functions as learning objectives. Similarly, other ranking-oriented CF methods can be classified into pairwise methods, listwise methods, and joint methods, according to the different types

[1] https://www.netflixprize.com/

of loss functions they adopted.

Rendle et al. [97] propose a popular pairwise ranking method, Bayesian personalized ranking (BPR), for implicit feedback datasets. Similar to MF models, BPR utilizes inner product as the scoring function, but optimizes a pairwise ranking loss function to train the model, for a given positive-negative feedback pair. Other pairwise methods such as [5, 96, 60, 24] also aim to learn the relative order between two items by minimizing various pairwise ranking loss functions. On the other hand, listwise methods such as [109, 131, 51] learn the scoring function by directly optimizing some ranking metrics (e.g., normalized discounted cumulative gain (NDCG)) defined on the predicted and actual item lists. Besides, joint methods [105, 65, 47] aim to model both absolute interests and relative preferences of users, by minimizing a joint objective function that consists of both pointwise and pairwise loss functions.

2.1.3 Neural Collaborative Filtering

Classical model-based CF methods (e.g., MF) are essentially linear models that are not able to learn the complicated interactions between users and items in large and complex recommender systems. To overcome this issue, some researchers generalize the idea of CF into deep learning and design a number of neural CF methods, which leverage some neural networks to model/predict the relevance scores between users and items, instead of the simple inner product of feature vectors in MF.

For instance, Wang et al. [133] propose a hierarchical Bayesian model, named collaborative deep learning (CDL), which jointly performs deep representation learning for the content information and CF for the feedback matrix. Wang et al. [132] develop a collaborative recurrent autoencoder (CRAE) which models the generation of content sequences in the CF setting. Zhang et al. [155] propose an integrated autoencoder-based framework, termed collaborative knowledge base embedding (CKE), which jointly learns the latent representations in CF and the items'

semantic representations from the knowledge base.

He et al. [42] propose a neural matrix factorization model, named NeuMF, which integrates multi-layer perceptron (MLP) with generalized matrix factorization to make predictions, and is trained by minimizing a binary cross-entropy loss over both observed positive user-item pairs and sampled negative ones. Song et al. [114] propose a similar neural network architecture to make predictions, but the network is trained with a different pairwise ranking loss function. Ebesu et al. [31] propose a collaborative memory network (CMN) method, which integrates memory networks with MF models to perform top-k recommendations. Liang et al. [70] propose a neural CF method based on variational autoencoders for implicit feedback recommendations, which utilizes a generative model with multinomial likelihood and uses Bayesian inference for parameter estimation. Recently, a number of neural CF methods that leverage graph convolutional networks (GCNs) have been proposed for a variety of recommendation domains and have demonstrated appealing performance [7, 91, 153, 142]. Besides, more works on neural CF methods can be found in a recent survey on deep learning based recommender systems [156].

2.1.4 Social Collaborative Filtering

Since traditional CF methods only exploit the user-item feedback data to discover users' preferences, their recommendation performance may be greatly reduced when data sparsity and cold-start (new user or new item problem) occur. Fortunately, with the emergence of online social networks such as the follower network in Twitter, the friend network in Facebook, and the trust network in Epinions, the additional social information of users is usually available to recommender systems. According to the social influence theory, people are influenced by their linked neighbors in the social network, leading to the homophily effect that social neighbors may share similar preferences with each other [9, 4, 122]. Thus, it is a potential way to improve the

performance of CF methods by effectively leveraging available social networks.

Social CF is a potential approach to alleviating the cold-start and data sparsity issues, which incorporates social networks into traditional CF methods, in order to help discover users' preferences and produce meaningful recommendations more effectively. A number of memory-based social CF methods [85, 52, 130] explore the trust propagation in social network, which generate rating predictions for a target user by directly aggregating the feedback data of his/her trusted friends. These memory-based social CF methods are able to improve the coverage of recommendations compared to traditional memory-based CF methods, but are not suitable to large-scale recommender systems as they need to compute similarities over the entire rating matrix and the whole social trust network.

Model-based social CF methods [82, 152, 107, 80, 53, 83, 81, 84, 13] employ the technique of MF to exploit the available social network data, which can be applied to large datasets. SoRec [82], TrustMF [152] and PSLF [107] factorize simultaneously the user-item rating matrix and the user-user social network simultaneously in a shared latent feature space. RSTE [80] fuses target user's interests and his/her trusted friends' tastes to model/predict a specific rating of a user-item pair. SocialMF [53] is similar to RSTE, but uses a different implementation to take into account the interests of trusted friends by incorporating a regularization term to control the distance between target user's feature vector and the averaged feature vector of his/her trusted friends. In [83], the authors introduce several social regularization terms that are similar to the one in SocialMF. The main difference is that the trust value in social regularization term is replaced by the Pearson correlation coefficient (PCC) calculated on users' rating data.

Recently, several complex social CF models beyond MF have been proposed for social recommender systems [74, 14, 149, 32, 17]. SREPS [74] simultaneously models the structural information in the social network, and the rating and consumption

information in the user-item feedback data under an essential preference space, by using both network embedding and MF. GraphRec [32] utilizes graph neural networks to learn user and item latent feature vectors from both user-item feedback graph and user-user social graph. SamWalker [17] simultaneously learns personalized data confidence and draws informative training instances by leveraging the social network information. SAMN [14] utilizes an attention-based memory module to learn user-friend relation vectors, and builds a friend-level attention component to adaptively select informative friends for user preference modeling. DANSER [149] introduces dual graph attention networks to collaboratively learn representations for two-fold social effects, which are captured by a user-specific attention weight and a dynamic context-aware attention weight, respectively.

2.1.5 Summary

In this section, we review the existing CF-based recommendation methods in the literature from a generalized perspective, including memory-based CF methods, MF models, ranking-oriented CF methods, neural CF models, and social CF methods. Despite their successes, these CF methods are originally designed for static single-step recommendation problems. Although some of them might be extended to the multi-step interactive recommendation problem by performing online updates to model parameters, they are still not able to provide satisfactory solutions since they are essentially supervised learning based methods that only aim to learn short-sighted recommendation models based on users' immediate feedback data. In spite of this, the practices of personalization and collaboration modeling in these prior works offer us valuable inspirations in designing our RL-based recommendation agents.

2.2 Reinforcement Learning

Reinforcement learning (RL) is a machine learning paradigm that focus on the problem of learning from interaction [119]. Different from supervised learning and unsupervised learning, RL aims to lean an agent that can auto-control its behavior in an environment, in order to achieve a goal. It has been proved that RL is able to effectively model long-term effects and make optimal multi-step decisions in many complex decision making tasks such as playing Atari games [90, 124, 144, 88, 125] and the game of Go [110, 113]. By applying RL to interactive recommendation, the recommendation agent is expected to model the long-term effects in the interactive recommendation process, and thus learn a farsighted policy to make desirable recommendations that achieve the maximal cumulative rewards.

2.2.1 Tabular Reinforcement Learning

In the literature, a typical RL-based recommendation approach contains two main stages. The first stage is to formulate the specific recommendation problem as an RL task, which is mathematically described by a Markov decision process (MDP) that consists of state space, action space, reward function, and state-transition function. The second stage is to develop a proper RL algorithm to solve the MDP, that is, to estimate the agent's optimal policy that maximizes the cumulative rewards.

Shani et al. [106] propose an MDP-based approach to book recommendation, which models the user clicked item sequence as the MDP's states and solves the MDP via dynamic programming. Unfortunately, for most of real-world recommender systems, the MDP is always unknown since both the state-transition function and reward function cannot be explicitly specified. Thus, model-based RL methods such as dynamic programming are not able to provide solutions in those cases. Instead, model-free RL methods do not require the specific dynamics of the environment's

model, and are able to update the agent's policy towards an optimal one by interacting with the environment in a trial-and-error fashion. Temporal-difference learning (e.g., Q-learning [145]) is one of the most popular model-free RL algorithms, which aims to estimate the optimal state-value or action-value function (corresponding to an optimal policy) by using bootstrapping [119]. Researchers have applied the temporal-difference based algorithms to some small-scale recommender systems. Taghipour et al. [120] propose a Q-learning based recommendation approach to a webpage recommender system. Silver et al. [112] propose a concurrent RL framework, which utilizes a variant of temporal-difference learning to learn policies efficiently from partial interaction sequences of users. Choi et al. [23] formulate the recommendation problem as a gridworld game by using a biclustering technique to convert the user-item matrix to a series of grid states, and estimate the optimal policy with some Q-learning based algorithms.

2.2.2 Deep Reinforcement Learning

In spite of their good convergence guarantees, the aforementioned tabular RL methods require maintaining a lookup table of all states or/and actions during the policy learning process. As a result, they are not able to handle the high-dimensional or continuous state and action spaces of most real-world recommender systems, and are of poor generalization abilities with respect to unseen states or actions. To overcome the weaknesses of tabular RL methods, an effective way is to leverage the technique of value function approximation to approximate the optimal action-value function (corresponding to an optimal policy) by using a parameterized function approximator [119, 72]. On the other hand, instead of using value function approximation, an alternative way is to directly parameterize the policy itself and update the parameterized policy towards an optimal one according to the policy gradient theorem [119]. Next, we first review the existing deep RL agents that utilize deep neural networks as

the function approximator to approximate the optimal action-value function (value-based deep RL) or the optimal policy (policy-based deep RL), followed by discussing some supervised learning techniques in deep RL-based recommender systems.

Value-based Deep RL. Deep Q-network (DQN) [90] is probably the most popular deep RL agent, which has demonstrated human-level or even better performance in playing the Atari games. Over the past few years, a lot of extensions or modifications to DQN such as Dueling DQN [144], Double DQN [124], and Noisy DQN [33] have been proposed for game-based RL tasks and have shown better performance. Due to the successes of DQN-based agents in game-based tasks, a number of researchers propose to incorporate the techniques of DQN into recommender systems and design a few DQN-based recommendation agents in different recommendation domains [160, 158, 19]. DEERS [158] is a DQN-based agent designed for product recommendation in e-commerce sites. It first utilizes gated recurrent units as the state representation module to learn high-level state representations from both the clicked and skipped item sequences of users, and then employ a parallel action-value prediction module to estimate action-values based on the two types of high-level state representations. DRN [160] applies the Dueling DQN agent to news recommendation, which estimates action-values based on the linear combination of state-value function and advantage function. Particularly, it relies on a manual state/action representation module that incorporates various hand-crafted features of users, news and contexts into the representations of states and actions. Robust DQN [19] extends the DDQN agent to online tip recommendation by proposing stratified sampling replay strategy and approximate regretted reward.

Policy-based Deep RL. Another major type of deep RL agents are based on the policy gradient theorem, such as Monte Carlo policy gradient (a.k.a. REINFORCE [146]) and actor-critic policy gradient [119]. Chen et al. [18] propose a REINFORCE-based deep RL agent to recommendation, by incorporating off-policy correction to

tackle the biases in the logged user feedback data collected from multiple behavior policies. Wang et al. [141] propose an action-critic policy gradient based deep RL agent, which utilizes recurrent neural networks to address the issue of partially observed states in real-world treatment recommender systems. Chen et al. [15] propose a tree-structured policy gradient recommendation (TPGR) framework, which builds a balanced hierarchical clustering tree over the items and formulates the item picking problem as path seeking from the root to a leaf of the tree. Moreover, deep deterministic policy gradient (DDPG) [71] is a popular actor-critic deep RL agent based on the deterministic policy gradient theorem [111], which is able to hand continuous action space and learn deterministic policies. A number of DDPG-based recommendation agents have been proposed to a variety of recommendation domains [157, 48, 76, 78].

Supervised Learning Techniques in Deep RL. A number of studies incorporate the techniques of supervised learning into deep RL-based recommender systems. Chen et al. [20] train a user behavior model based on offline logged data by leveraging generative adversarial networks. The trained user behavior model is treated as an environment simulator, which is used to interact the agent and promote the learning of RL-based recommendation policies. Shi et al. [108] build a similar environment simulator with generative adversarial networks based on the logged data in a large-scale e-commerce website. Zou et al. [163] propose a Pseudo Dyna-Q framework, which iteratively learns an environment simulator and a DQN agent from the logged data. Besides, Liu et al. [75] propose a supervised learning-based user/item embedding component for deep RL-based recommendation agents, in order to learn more effective high-level state/action representations.

2.2.3 Summary

In this section, we review the existing RL-based recommendation methods in the literature, including tabular RL methods and deep RL methods. The tabular RL

methods were proposed very early, but did not attract much attention in the research community, since they are not able to deal with the large state and action spaces in modern recommender systems and have poor generalization ability. In contrast, the recently proposed deep RL methods have drawn more and more attention, due to their generalization ability, applicability, and impressive recommendation performance of course. However, the existing deep RL methods fail to effectively model the key factors of personalization and collaboration in real-world recommender systems, which cannot produce a truly personalized recommendation policy. Different from the prior works, in this thesis we systematically investigate the problem of modeling personalization and collaboration under the framework of deep RL, and develop several truly personalized deep RL agents for interactive recommendation from different perspectives. Another main difference between the existing methods and ours is that most of them are designed for some special recommendation domains (e.g., news, video, or e-commerce), while our proposed methods rely only on user-item feedback data, which are applicable to almost all types of recommender systems. Moreover, most of the existing techniques and ours are complementary, which can be combined to further improve the performance of deep RL-based recommendations.

Chapter 3

User-specific Deep Q-network: Modeling Personalization and Collaboration via MF-based Representation Module

Existing RL-based recommendation agents are significantly limited by a common weakness that the key factors of personalization and collaboration are not effectively modeled. In this work, we propose a novel way to model the factors of personalization and collaboration into deep RL agents. We first construct user-specific latent vector representations of states and actions by employing matrix factorization (MF), a well adopted collaborative filtering (CF) technique in traditional recommendation problems. After that, we propose User-specific Deep Q-network (UDQN) to approximate the optimal action-value function (corresponding to an optimal recommendation policy) based on the constructed latent representations using Q-learning. Finally, we validate the effectiveness of our approach with comprehensive experimental results and analysis on both explicit-feedback and implicit-feedback recommendation tasks.

3.1 Introduction

Personalized recommender systems have been successfully applied in many web applications, such as E-commerce sites, social networking platforms, and music/movie sites. While such recommender systems are intrinsically interactive, only a few of interactive recommendation approaches are available in the literature [2, 159, 143]. Most of existing works aim at how to build accurate models based on user history data for next-step recommendations [1, 59, 58, 100]. The "interactive" recommendations are insufficiently implemented by performing online updates to traditional recommendation models [29, 93, 43]. However, in real-world scenarios, the recommender usually interacts with the user for multiple steps, and the recommendations provided at current step may affect the quality of future recommendations. For example, the items recommended in earlier steps may not be liked by the user, but the received feedbacks provide useful information which can help the recommender make better recommendations in later steps.

In the literature, a number of studies have focused on interactive recommendations or other related problems. Some researchers incorporate the multi-armed context-free or contextual bandit algorithms into recommendation approaches for recommending movies [159, 55], music [143], restaurants [25], news articles [69, 126, 123], and other domains [134, 147, 135, 148]. They model the recommendation problem as a repeated game of arm selection [8, 129], with emphasis on the exploitation-exploration dilemma [3, 35]. Unfortunately, these bandit-based methods ignore the important influences from earlier steps to later steps, and cannot provide satisfactory solutions to the multi-step interactive recommendation problem in practice.

To address this issue, a potential solution is to utilize the techniques of reinforcement learning (RL) [119] to model the multi-step recommendations as a sequential decision making problem. It has been proved that RL has the ability to make op-

timal multi-step decisions for many complex problems such as playing Atari [90] and the game of Go [110]. By using RL, an intelligent agent can be learned to recommend an appropriate item at each step, so as to maximize the globally optimal recommendation performance for all steps. In the literature, RL based approaches have been proposed to solve a number of recommendation problems such as session-based recommendation [106, 157, 158], news article recommendation [160], and music playlist recommendation [46]. However, these approaches are only designed for implicit-feedback recommender systems, which cannot handle explicit-feedback recommendation problems. Moreover, they fail to model the information of personalized preferences and collaborative relationships, leading to inefficiency in providing high-quality personalized recommendations for diverse users.

Distinct from the existing works, we study an interactive recommendation problem for general recommender systems, and propose a user-specific RL based approach which well models users' preferences and relationships. Specifically, we first formulate the problem of T-step interactive recommendation for each target user as a Markov decision process (MDP). However, from the practical perspective, it is infeasible to estimate the optimal policy for each user's MDP due to limited data and high computation cost. Thus, we suggest to estimate a global policy for all users' MDPs, leading to a non-typical multi-MDP RL task. Compared to traditional tasks, it is a more difficult challenge to learn the optimal policy for such multi-MDP task, as the MDPs of different users may vary remarkably in state transitions. To handle this challenge, we construct user-specific latent states to connect different MDPs by using the technique of matrix factorization. After that, we propose a DQN [90] based learning method, to estimate optimal policies based on the constructed user-specific latent states, which is referred to as User-specific DQN (UDQN). Furthermore, we propose a Biased UDQN (BUDQN) method to explicitly model user-specific information of preferences by employing an additional bias parameter to capture the differences in

different users' Q-values. We empirically validate our approach to interactive recommendation on real-world datasets. The experimental results demonstrate that it achieves significant improvements over the state-of-the-art approaches.

3.2 Preliminaries

3.2.1 Problem Definition

Suppose we have a 5-star recommender system with integer ratings in $\{1, 2, 3, 4, 5\}$, which currently involves m users and n items. Let $\mathcal{U} = \{1, ..., m\}$ and $\mathcal{I} = \{1, ..., n\}$ denote the sets of users and items, respectively. Let $R \in \mathbb{R}^{m \times n}$ denotes the observed user-item rating matrix, where each nonzero R_{ui} denotes the observed rating of item i given by user u, and each zero implies that the user has not rated the item yet. We consider a cold-start recommendation scenario as follows. Suppose a target user $u = m + 1$ enters into the system at time step $t = 0$. The recommender provides an item to user u, then receives a rating on the item given by him/her. After considering the observed rating, the recommender updates its knowledge about user u and provides a new item at time step $t = 1$. Suppose such a recommender-user interactive process will last for T time steps. The goal of the recommender is to provide user u with the most interesting items that maximize the sum (or average) of ratings received over T steps.

3.2.2 Q-learning

In RL, an agent interacts with an environment and learns a policy to maximize the cumulative reward it receives. Normally, the policy is a mapping from states to actions, $a = \pi(s)$, or probability distributions over actions given states, $\pi(a|s)$. To learn the optimal policy, the basic idea of many reinforcement learning methods is

to estimate the action-value (Q) function. The Q function is formally defined as:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum\nolimits_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right], \tag{3.1}$$

where γ is the discount factor that balances the importance between future rewards and immediate rewards. It indicates, in the long run, how good it is to take action a in state s while following policy π in future steps. The optimal Q function, $Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$, obeys an important identity known as the Bellman optimality equation [6]:

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') | s, a\right]. \tag{3.2}$$

Based on the Bellman optimality equation, Q-learning [145] estimates a Q function (i.e., a lookup table) via the following online update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right], \tag{3.3}$$

where α is the learning rate, and (s,a,r,s') denotes a transition the agent experiences during its interactions with the environment. The estimated Q function will converge to Q^* when the number of iterations goes to infinity [119]. The greedy policy w.r.t. Q^* will be an optimal policy π^*. The classical Q-learning algorithm must maintain a lookup table of all state-action pairs, which cannot handle complex reinforcement learning tasks with enormous state and action spaces. Moreover, it estimates the action-value function separately for each sequence, which has no generalization ability to deal with unseen states and actions [119].

To address these issues, a common practice is to estimate the action-value function by using a function approximator, i.e., $\hat{Q} \approx Q^*$. For instance, Deep Q-learning (DQN) [89, 90] employs a deep neural network with weights $\mathbf{w}$, referred to as Q-network $\hat{Q}(s,a,\mathbf{w})$, as the function approximator. To update $\hat{Q}$, one can minimize a squared loss function defined as:

$$\mathcal{L} = \sum\nolimits_{s,a,r,s'}\left(r + \gamma \max_{a'} \bar{Q}(s',a') - \hat{Q}(s,a,\mathbf{w})\right)^2, \tag{3.4}$$

where $\bar{Q}$ denotes a target network which copies the weights of $\hat{Q}$ regularly after L steps during the interactive process. In practice, rather than directly optimizing the above loss function, a more convenient way is to perform stochastic gradient descent (SGD) on a sampled transition (s, a, r, s') [89, 90]:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[r + \gamma \max_{a'} \bar{Q}(s', a') - \hat{Q}(s, a, \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{Q}(s, a, \mathbf{w}). \tag{3.5}$$

3.3 User-specific Deep Q-network

3.3.1 The Markov Decision Process (MDP)

We consider the aforementioned recommendation problem under the standard RL framework. The recommender-user interaction in recommendation can be naturally modeled as the agent-environment interaction in RL (see Figure 1.1). At each time step t, the agent (recommender) observes a state s_t about the environment (user u), then takes an action (item) a_t according to its policy π, which is usually a mapping from states to action probabilities. One time step later, as a result of its action, the agent receives a numerical reward (rating) r_{t+1} and a new state s_{t+1} from the environment. The goal of the agent is to maximize the cumulative reward it receives over T steps. According to [119], the environment can be mathematically described by a Markov decision process (MDP), a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ defined as follows.

$\mathcal{S}$ is the state space. The state s_t represents the observed preferences of user u at time step t. A straightforward state representation method is to define the state s_t as a n-dimensional rating vector $R_{u*}^{(t)}$, which denotes the u-th row of R at time step t. The nonzero values of $R_{u*}^{(t)}$ indicate the observed ratings given by user u. Obviously, the initial state s_0 is a zero vector.

$\mathcal{A}$ is the action space. We define $\mathcal{A}$ as the set of all items, i.e., $\mathcal{A} = \mathcal{I}$. In each state s_t, an action a_t can be taken from the set of available actions $\mathcal{A}(s_t)$, which is defined recursively: $\mathcal{A}(s_t) = \mathcal{A}(s_{t-1}) \backslash \{a_{t-1}\}$ for $t \neq 0$, and $\mathcal{A}(s_0) = \mathcal{A}$. In other

words, the agent is not allowed to choose the items that have been recommended at previous time steps.

$\mathcal{P}$ is the transition probability. $\mathcal{P}^a_{ss'} = Pr[s_{t+1} = s' | s_t = s, a_t = a]$ denotes the probability that the environment transits to state s' after receiving action a in state s. In the recommendation setting, the exact transition probabilities are unknown in advance. The agent can observe specific state transitions by interacting with the environment step by step.

$\mathcal{R}$ is the reward function. $\mathcal{R}^a_{ss'} = E[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s']$ denotes the expected immediate reward the environment generates after the transition from state s to s' due to action a. In the recommendation setting, the immediate reward of executing an action a only depends on the rating given by user u. Therefore, we define $\mathcal{R}^a_{ss'} = R_{ua}$.

Note that the above MDP has the **Markov property**:

$$\begin{aligned} & Pr[s_{t+1} = s', r_{t+1} = r | s_0, a_0, r_1, ..., s_{t-1}, a_{t-1}, r_t, s_t, a_t] \\ & = Pr[s_{t+1} = s', r_{t+1} = r | s_t, a_t], \end{aligned} \tag{3.6}$$

for all s', r, and histories $s_0, a_0, r_1, ..., s_{t-1}, a_{t-1}, r_t, s_t, a_t$.

3.3.2 The Multi-MDP Reinforcement Learning Task

Once the MDP for a target user u is given, it seems straightforward to apply standard RL algorithms to learn the agent's optimal policy, then use the learned policy to make recommendations. Unfortunately, this is far from the desirable solution that can really work in practice. In traditional RL tasks such as Atari games [90], the agent can interact with a single MDP (i.e., environment) continuously and obtain any amount of experience to update its policy towards an optimal one. In recommendation settings, however, it is practically infeasible to interact with a single user's MDP repeatedly to obtain so much experience. Even if there is such a user, the learned policy cannot

provide valuable and novel recommendations due to over-fitting. Moreover, learning an independent policy for each user is time-consuming, and does not consider the possible relationships between different users, which are very important to discovering users' unknown interests in real-world recommender systems.

As such, the agent must interact with the MDPs of all involved users, and learn a global recommendation policy based on them. This actually gives rise to a multi-MDP RL task, as shown in Figure 1.2. Undoubtedly, compared to traditional tasks, it is a more difficult challenge to estimate the agent's optimal policy for the multi-MDP task. Due to the different tastes of users, their MDPs vary remarkably with distinctly different state transitions. Similar users may have relatively close MDPs, but dissimilar users may have totally distinct ones. The experience collected from different MDPs may be diverse and inconsistent, which makes it very hard to estimate the optimal policy. Thus, the key challenge of the multi-MDP task is how to effectively model the possible relationships (including similarities and differences) between different MDPs.

3.3.3 User-specific Latent States based on Matrix Factorization

To address the aforementioned challenge, we employ matrix factorization (MF) to convert the MDPs' raw states s_t^o (as defined in Section 3.3.1) to user-specific latent states. It has been proved that MF has powerful capability in modeling both the users' preferences and the possible relationships between different users [59]. By mapping all users and items into the shared low-dimensional vector space, some effective latent features can be constructed to represent users' preferences and relationships. Thus, it is natural and reasonable to utilize the latent feature vector of target user u to represent the MDP(u)'s states.

More specifically, we first pre-train a MF model based on the observed rating

matrix $R \in \mathbb{R}^{m \times n}$, by using stochastic gradient descent (SGD) to minimize a squared loss function defined as:

$$\mathcal{L} = \sum\nolimits_{(u,i) \in \Omega} (U_u^{\mathrm{T}} V_i - R_{ui})^2 + \lambda(\|U\|_{\mathrm{F}}^2 + \|V\|_{\mathrm{F}}^2), \tag{3.7}$$

where $\Omega = \{(u, i) : R_{ui} \neq 0\}$ denotes the set of observed user-item pairs, $\| \cdot \|_{\mathrm{F}}$ denotes the Frobenius norm, λ is the regularization parameter, and $U \in \mathbb{R}^{d \times m}$ and $V \in \mathbb{R}^{d \times n}$ denote the latent feature matrices of users and items, respectively. In each iteration, we traverse the observed user-item pairs in Ω. The update rule of SGD for a user-item pair (u, i) is given by:

$$\begin{aligned} U_u &\leftarrow U_u - 2\alpha \left[(U_u^{\mathrm{T}} V_i - R_{ui}) V_i + \lambda U_u \right] \\ V_i &\leftarrow V_i - 2\alpha \left[(U_u^{\mathrm{T}} V_i - R_{ui}) U_u + \lambda V_i \right], \end{aligned} \tag{3.8}$$

where α is the learning rate. The time complexity of pre-training is $O(cd|\Omega|)$, where c is the number of iterations, d is the dimensionality of latent space, and $|\Omega|$ is the number of observed user-item pairs. As both c and d are usually small constants (we set $c = 20$ and $d = 64$ in our experiments) in practice, the cost of pre-training is quite low, which is linear to the size of observed rating data. Note that we tested $c \in \{10, 20, 30\}$, but the results did not show significant differences.

The pre-trained item feature vectors $V_i \in \mathbb{R}^d$ for all items $i = 1, ..., n$, will be fixed and used to update the user feature vector $U_u \in \mathbb{R}^d$ for target user u. During the agent-environment interactive process, we continuously maintain the user feature vector U_u over time steps, and use it as the latent state s_t. Moreover, to ensure efficient online learning, U_u is updated based on the latest observed rating R_{ui} (which is located in the raw state s_t^o), according to Equation (6.17). The detailed algorithm of constructing latent states is presented in Algorithm 3.1. Similar to the online updates in [43], for the While loop, one iteration is usually sufficient to achieve good results.

Algorithm 3.1: Constructing User-specific Latent States by Matrix Factorization

Input: raw state s_t^o, vector U_u, pre-trained V, learning rate α, regularization λ, time step t

Output: latent state s_t, vector U_u

1 **if** $t = 0$ **then**

2 Initialize U_u with zeros

3 **else**

4 **while** $\mathcal{L}$ *is not converged* **do**

5 **Update U_u according to Equation (3.8)**

6 $s_t \leftarrow U_u$

With the MF-based latent states, we actually derive a set of MDPs with a low-dimensional continuous state space. Note that the MF-based MDPs still have Markov property, as state s_t is updated based on only state s_{t-1}, which is independent of $s_{t-2}, s_{t-3}, ..., s_0$. This indicates that we can still employ standard reinforcement learning methods to estimate the optimal policy. Besides, other more complex MF models such as those proposed in [59] can be easily incorporated into the framework to construct latent states. The performance is supposed to be consistently improved as long as the adopted models can better capture the users' preferences and relationships. However, in this paper, we only use the standard MF model as an instantiation due to its simplicity, and focus on validating the effectiveness of the general framework of our proposed approach.

3.3.4 The UDQN Model

To learn the optimal policy based on user-specific latent states, we propose a DQN [90] based method, named User-specific Deep Q-network (UDQN). Specifically, we employ a feedforward neural network as our Q-network $\hat{Q}(s, a, \mathbf{w})$. The Q-network $\hat{Q}$ uses the latent state s_t as input, and outputs the Q values of all possible actions in that state. To update our Q-network, we use the same update rule in Equation (3.5). The basic framework and the detailed algorithm of UDQN are presented in

Figure 3.1 and Algorithm 3.2, respectively.

To balance the MDPs of different users, we utilize a uniform sampling scheme to select transitions for estimating Q-values. In each episode of the interactive process, a user u is uniformly sampled from training set $\mathcal{U}_{train}$ to construct the current environment. It will be used to interact with the agent for T time steps, and to generate corresponding T transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ for $t = 0, ..., T-1$ based on user u's data. At each time step t, the agent observes reward r_{t+1} and raw state s^o_{t+1} from the environment after executing action a_t. Then, the latent state s_{t+1} is computed by Algorithm 3.1, and the transition $(s_t, a_t, r_{t+1}, s_{t+1})$ is added into an experience replay memory $\mathcal{M}$. When performing Q-learning updates, in stead of single transition, a minibatch of transitions is uniformly sampled from $\mathcal{M}$ to update the Q-network. Moreover, to ensure exploration at each time step t, the action a_t is chosen by using a ϵ-greedy strategy with regard to the predicted Q-values. The training process can continue for any number of episodes as long as the Q-network is not converged. After training, the learned Q-network $\hat{Q}$ can be used to make T-step interactive recommendations for any new user. The agent only needs to interact with the user step by step, observe states, and always take greedy actions with respect to the Q-values outputted by $\hat{Q}$.

Biased UDQN

In the proposed UDQN method, the user-specific information of each user u is only implicitly embedded in the latent state s^u of MDP(u), and the Q-values for all users' states are estimated by a unified Q-network $\hat{Q}$. This method can be further improved in order to perform more effective recommendations in real-world systems, since many users may have significantly different Q-functions due to diverse tastes and behaviors. For example, during the interactive recommendation process, user u may give much higher ratings (i.e., immediate rewards) on the recommended items

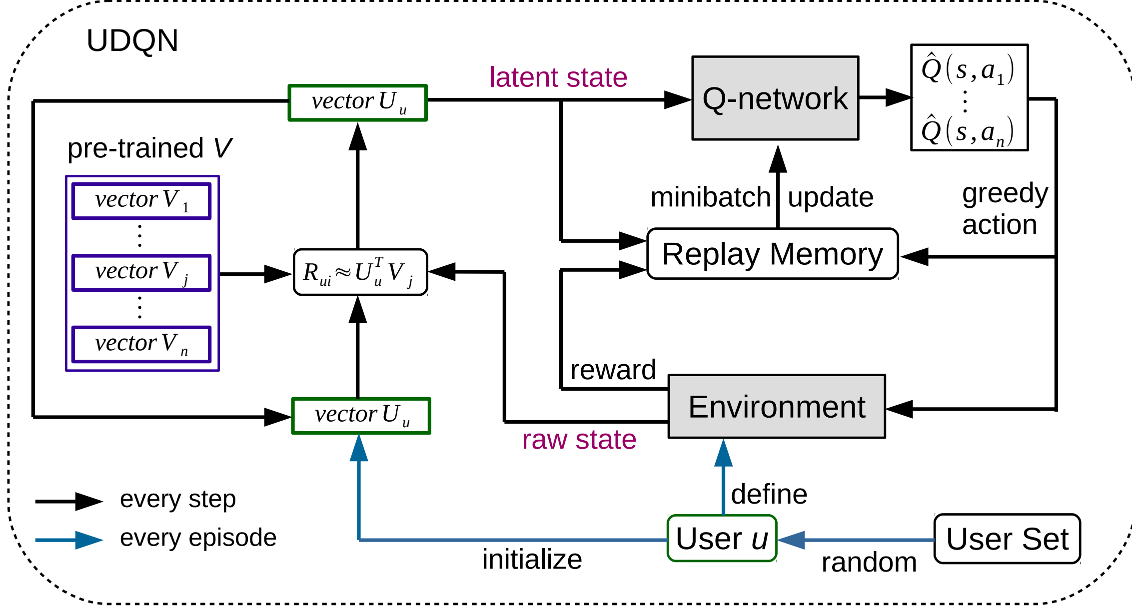


Figure 3.1: The basic framework of UDQN.

compared to user w. In such cases, the Q-values (i.e., long-term rewards) of user u's states $\{s_1^u, s_2^u, s_3^u, ...\}$ will be much larger than the Q-values of user w's states $\{s_1^w, s_2^w, s_3^w, ...\}$, for all actions (i.e., items). However, the latent states with a unified Q-network are not enough for modeling such differences.

Inspired by the Biased MF models [59], we propose a novel method, named Biased UDQN (BUDQN), to explicitly model the differences between users by adding a simple bias parameter $\mathbf{b}_u$ into the Q-function of target user u. More specifically, we redefine the value $Q(s_t^u, a_t)$ as:

$$Q(s_t^u, a_t) = \mathbf{b}_u + \hat{Q}(s_t^u, a_t, \mathbf{w}), \tag{3.9}$$

where $\hat{Q}$ denotes the same Q-network of UDQN. Accordingly, the loss function is redefined as:

$$\mathcal{L} = \sum\nolimits_{s_t^u, a_t, r_{t+1}, s_{t+1}^u} \left(r_{t+1} + \gamma \max_a \bar{Q}(s_{t+1}^u, a) - Q(s_t^u, a_t)\right)^2, \tag{3.10}$$

where $(s_t^u, a_t, r_{t+1}, s_{t+1}^u)$ denotes a transition sampled from MDP(u), and $\bar{Q}(s_{t+1}^u, a) = \bar{\mathbf{b}}_u + \bar{Q}(s_{t+1}^u, a, \bar{\mathbf{w}})$ regularly copies Q's parameters $\mathbf{b}_u$ and $\mathbf{w}$ every L steps during the

Algorithm 3.2: The Learning Algorithm of UDQN

Input: training set $\mathcal{U}_{train}$, rating data R, the number of episodes K, the number of time steps T, discount factor γ, ϵ-greedy parameter ϵ, replay memory $\mathcal{M}$

Output: the learned Q-network $\hat{Q}$

1 Initialize $\hat{Q}$ with random weights
2 **for** *episode* $= 1, ..., K$ **do**
3 Uniformly pick a user $u \in \mathcal{U}_{train}$ to construct the current environment
4 Observe raw state s_0^o
5 Compute latent state s_0 by Algorithm 3.1
6 **for** $t = 0, ..., T-1$ **do**
7 Select action a_t using ϵ-greedy policy w.r.t. $\hat{Q}$
8 Execute action a_t, observe reward r_{t+1} and raw state s_{t+1}^o
9 Compute latent state s_{t+1} by Algorithm 3.1
10 Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{M}$
11 Sample a minibatch of (s, a, r, s') from $\mathcal{M}$
12 Update $\hat{Q}$ according to Equation (3.5)

interactive process, which is held fixed when optimizing the loss function. Specifically, the gradient of $\mathcal{L}$ with respect to $\mathbf{w}$ and $\mathbf{b}_u$ is given by:

$$\begin{aligned}
\nabla_{\mathbf{w}}\mathcal{L} =& -2\sum\nolimits_{s_t^u, a_t, r_{t+1}, s_{t+1}^u} \left[r_{t+1} + \gamma \max_a \bar{Q}(s_{t+1}^u, a) - Q(s_t^u, a_t)\right] \nabla_{\mathbf{w}} Q(s_t^u, a_t) \\
=& -2\sum\nolimits_{s_t^u, a_t, r_{t+1}, s_{t+1}^u} \left[r_{t+1} + \gamma \max_a \bar{Q}(s_{t+1}^u, a) - Q(s_t^u, a_t)\right] \nabla_{\mathbf{w}} \hat{Q}(s_t^u, a_t, \mathbf{w}), \\
\nabla_{\mathbf{b}_u}\mathcal{L} =& -2\sum\nolimits_{s_t^u, a_t, r_{t+1}, s_{t+1}^u} \left[r_{t+1} + \gamma \max_a \bar{Q}(s_{t+1}^u, a) - Q(s_t^u, a_t)\right] \nabla_{\mathbf{b}_u} Q(s_t^u, a_t) \\
=& -2\sum\nolimits_{s_t^u, a_t, r_{t+1}, s_{t+1}^u} \left[r_{t+1} + \gamma \max_a \bar{Q}(s_{t+1}^u, a) - Q(s_t^u, a_t)\right].
\end{aligned} \tag{3.11}$$

Therefore, the update rule of SGD on a given transition $(s_t^u, a_t, r_{t+1}, s_{t+1}^u)$ is:

$$\begin{aligned}
\mathbf{w} \leftarrow& \mathbf{w} + \alpha \left[r_{t+1} + \gamma \max_a \bar{Q}(s_{t+1}^u, a) - Q(s_t^u, a_t)\right] \nabla_{\mathbf{w}} \hat{Q}(s_t^u, a_t, \mathbf{w}), \\
\mathbf{b}_u \leftarrow& \mathbf{b}_u + \alpha \left[r_{t+1} + \gamma \max_a \bar{Q}(s_{t+1}^u, a) - Q(s_t^u, a_t)\right].
\end{aligned} \tag{3.12}$$

The learning algorithm of BUDQN can be derived by using the above update rule to replace the one in Algorithm 3.2. Compared to UDQN, the BUDQN method is more

powerful and flexible in estimating the optimal recommendation policy for multiple users.

Computational Complexity Analysis

We now analyze the time complexity of UDQN (Algorithm 3.2). In the inner For loop, the time is mainly taken by computing Q-values, computing latent state s_{t+1}, and updating $\hat{Q}$. The costs of computing Q-values and updating $\hat{Q}$ are both $O(|\mathbf{w}|)$, where $|\mathbf{w}|$ denotes the number of $\hat{Q}$'s weights. The cost of computing latent state is $O(d)$ (since only one iteration is needed for the While loop in Algorithm 3.1), where d is the dimensionality of latent feature space. Therefore, the time complexity of UDQN is $O(KT(d + |\mathbf{w}|))$, where K is the number of episodes and T is the number of time steps. Similarly, one can derive that the time complexity of BUDQN is also $O(KT(d + |\mathbf{w}|))$.

3.4 Experiments on Explicit Feedback Recommendation Tasks

3.4.1 Experimental Settings

Datasets

We use three benchmark rating datasets for conducting experiments, including two MovieLens datasets: ML100K and ML1M[1], one Yahoo! music dataset: YMusic[2]. The three datasets contain explicit ratings of items given by users. For YMusic, we remove the users who have fewer than 20 ratings, so as to be consistent with the MovieLens datasets, as well as to ensure there is enough data for training and testing. The statistics of datasets are presented in Table 3.1.

[1] http://grouplens.org/datasets/movielens/

[2] http://webscope.sandbox.yahoo.com/catalog.php?datatype=r

Table 3.1: The statistics of datasets

Statistics	**ML100K**	**ML1M**	**YMusic**
#users	943	6,040	5,050
#items	1,682	3,952	1,000
#ratings	100,000	1,000,209	174,497
rating scale	1-5	1-5	1-5
avg. rating	3.529	3.581	2.799

Evaluation Protocol

To evaluate interactive recommendation algorithms, we follow an unbiased offline evaluation scheme suggested in previous work [159, 69] that the pre-collected ratings are treated as unbiased interactive feedback of users. We regard those users who have more than 100 ratings as candidates for testing purpose. For each dataset, we split the data by randomly choosing 10% candidates as testing set $\mathcal{U}_{test}$, and the remaining as training set $\mathcal{U}_{train} = \mathcal{U} \backslash \mathcal{U}_{test}$. We repeat the above process 5 times independently and obtain 5 data splits. We conduct each experiment based on the 5 data splits and report the average results for evaluation. The evaluation metric we used is the average reward (rating) received over T time steps.

Moreover, how to regard the unknown (missing) ratings is non-trivial for evaluating recommendation algorithms. In the literature, there are two widely-used strategies. The unknown ratings are usually ignored in rating prediction task [59], and regarded as negative feedback in top-N recommendation task [26]. To comprehensively evaluate the algorithms, we adopt both of the strategies and derive two different tasks for T-step interactive recommendation:

- **Task I**. We ignore the unknown ratings. The possible actions (items) available for the agent are restricted in the set of rated items of the target user.
- **Task II**. We regard the unknown ratings as negative feedback. The entire

item set is available for the agent. The reward of recommending an item with unknown rating is defined as 0.

Using movie recommendation as an example, Task I focuses on predicting how much the user will like a movie, while Task II aims at predicting both whether the user will watch a movie, and how much she will like it. In general, Task II is more difficult than Task I.

Baselines

We compare our methods UDQN and BUDQN against a wide variety of baselines:

- **Random**. A simple method that picks items randomly.
- **Popular**. A simple method that picks the most popular items.
- **SVD++** [57]. A state-of-the-art matrix factorization (MF) method.
- **ICF** [159]. A state-of-the-art context-free ϵ-greedy bandit algorithm based on both observed and latent features.
- **LinUCB** [69]. A state-of-the-art contextual bandit method for news recommendation, which is extended for our problem by using the same learning framework of UDQN, and by concatenating U_u with V_i as its context vector.
- **DQN** [90]. A state-of-the-art Deep Q-learning algorithm, which estimates Q-values based on the Raw-based states (as described in Section 3.3.1).

In addition, we compare a variant of UDQN, called UDQN2, which utilizes user feature vector U_u and item feature vector V_a to represent state s and action a, respectively. The Q-network of UDQN2 takes the concatenation of U_u and V_a as input, and outputs the value of $Q(s, a)$. The purpose of comparing UDQN2 is to validate how our approach behaves with different types of Q-networks.

Parameter Settings

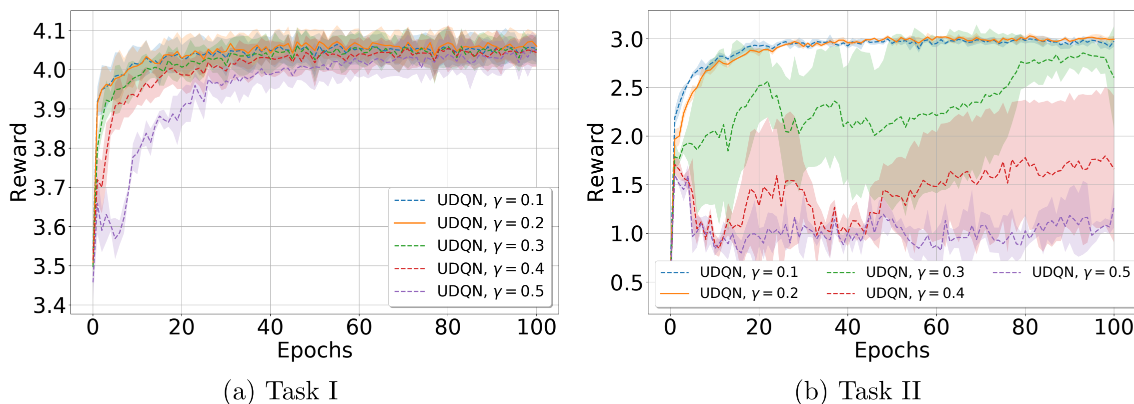
We set the hyperparameters of compared methods based on the cross validation on ML100K dataset. For reinforcement learning based methods, we use a two hidden-layer Q-network, where each consists of 256 units, followed by a rectifier nonlinearity. We set the discount factor $\gamma = 0.2$, the ϵ-greedy parameter $\epsilon = 0.1$, the minibatch size $|batch| = 1$, the replay memory size $|\mathcal{M}| = 1$, the target network parameter $L = 1000$, the learning rate $\alpha = 0.0001$ (for updating Q-networks), and the number of time steps in training phase $T_{train} = N(u)/2$ for user u, where $N(u)$ denotes the number of observed ratings of user u in the dataset. For the MF models, we set the dimensionality of latent space $d = 64$, the regularization parameter $\lambda = 0.01$, and the learning rate $\alpha = 0.01$ (for updating feature vectors). For LinUCB, we set the parameter $\alpha = 0.5$.

3.4.2 Performance Comparison

Results on Task I

We first compare the performance of all methods on Task I. The mean and standard deviation of the results over 5 runs on all datasets for both $T = 30$ and $T = 50$ are reported in Table 3.2. In each case, the bold font indicates the best performed method among all, while the mark $*$ denotes the best performed baseline. The relative improvement of those two methods is shown in the last row. From the results in Table 3.2, we have the following findings.

First of all, the proposed UDQN, UDQN2 and BUDQN significantly outperform the baselines in all cases, especially in comparison to DQN which use Raw-based states as input. This highlights the benefits of using the user-specific latent states to estimate Q-values. Secondly, BUDQN shows consistent and remarkable improvement over UDQN in all cases, which proves that the explicit user-specific information can



(a) Task I (b) Task II

Figure 3.2: Effect of the discount factor γ on the performance of UDQN.

further improve the Q-learning performance. Thirdly, the performance of UDQN2 has no significant difference with UDQN, which demonstrates that our approach is robust and insensitive to the two different types of Q-networks. Moreover, we also find that the MF model SVD++ performs relative better than other baselines in most cases. This is reasonable since Task I is related to the Rating Prediction tasks, in which the MF models have shown dominant advantages over others [57, 59].

Results on Task II

We now compare the performance of all methods on Task II. The mean and standard deviation of the results over 5 runs on all datasets for both $T = 30$ and $T = 50$ are reported in Table 3.3. The results on Task II show similar trends to Task I. First of all, the proposed UDQN, UDQN2 and BUDQN remarkably outperform the competitors in all cases again. Secondly, BUDQN is still the best performed method, whose improvements over the best performed baselines are at least 8.04% (on YMusic dataset for $T = 50$) and at most 23.99% (on ML1M dataset for $T = 50$). Thirdly, UDQN and UDQN2 still show very close performance. Moreover, the Popular method performs relatively better than other baselines.

Table 3.2: Performance comparison on Task I.

Method	**T=30**			**T=50**		
	ML100K	**ML1M**	**YMusic**	**ML100K**	**ML1M**	**YMusic**
Random	3.471±0.062	3.607±0.018	2.869±0.023	3.488±0.039	3.601±0.017	2.897±0.050
Popular	3.836±0.070	4.070±0.016	3.085±0.030	3.826±0.068	3.983±0.018	3.001±0.041
SVD++	4.104±0.070	4.294±0.019*	3.264±0.099	4.029±0.060*	4.215±0.014*	3.232±0.112
ICF	4.094±0.046	4.219±0.016	3.335±0.035*	4.024±0.057	4.173±0.013	3.262±0.053*
LinUCB	4.109±0.063*	4.256±0.018	3.182±0.053	4.015±0.061	4.163±0.014	3.123±0.070
DQN	4.058±0.050	4.267±0.020	3.149±0.109	3.989±0.047	4.195±0.014	3.066±0.094
UDQN	4.166±0.062	4.331±0.015	3.428±0.089	4.076±0.039	4.246±0.016	3.297±0.108
UDQN2	4.161±0.055	4.325±0.016	3.437±0.096	4.066±0.047	4.233±0.015	3.309±0.103
BUDQN	**4.202±0.048**	**4.362±0.017**	**3.504±0.110**	**4.114±0.034**	**4.275±0.016**	**3.343±0.104**
Improve	2.27%	1.59%	5.08%	2.10%	1.42%	2.50%

Table 3.3: Performance comparison on Task II.

Method	**T=30**			**T=50**		
	ML100K	**ML1M**	**YMusic**	**ML100K**	**ML1M**	**YMusic**
Random	0.467±0.037	0.270±0.012	0.510±0.218	0.466±0.017	0.271±0.008	0.510±0.230
Popular	2.617±0.014	2.537±0.040*	1.903±0.323*	2.429±0.031	2.292±0.034*	1.680±0.277*
SVD++	1.633±0.040	1.292±0.003	0.701±0.193	1.585±0.049	1.459±0.032	0.828±0.187
ICF	2.249±0.133	1.137±0.049	1.107±0.201	2.174±0.085	1.279±0.039	1.062±0.277
LinUCB	2.731±0.022*	2.477±0.032	1.761±0.330	2.460±0.024	2.239±0.033	1.355±0.276
DQN	2.706±0.072	2.287±0.040	1.460±0.298	2.544±0.010*	2.159±0.050	1.251±0.202
UDQN	3.250±0.035	2.974±0.064	2.156±0.259	3.015±0.040	2.801±0.057	1.789±0.211
UDQN2	3.224±0.025	2.956±0.052	2.173±0.220	3.010±0.055	2.785±0.049	1.793±0.240
BUDQN	**3.309±0.022**	**3.014±0.069**	**2.217±0.242**	**3.086±0.007**	**2.842±0.062**	**1.815±0.246**
Improve	21.14%	18.80%	16.50%	21.31%	23.99%	8.04%

3.4.3 Parameter Analysis

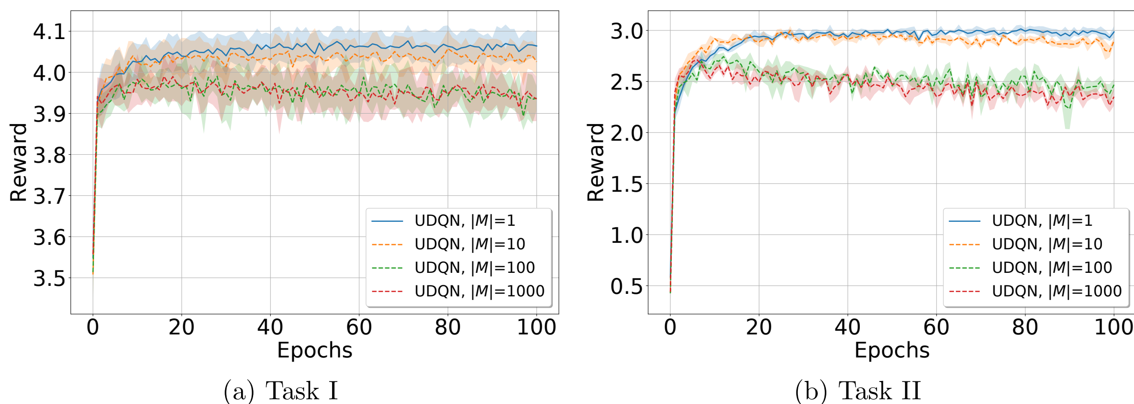
In this subsection, we show how our method UDQN behaves with different settings of some important parameters. When analyzing a specific parameter, others are fixed to the default settings. This set of experiments is conducted on ML100K dataset for $T = 50$.

Effect of the Discount Factor γ

The discount factor $\gamma \in [0, 1]$ balances the trade-off between future rewards and immediate rewards when estimating Q-values. The selection of its value usually depends on the natures of particular reinforcement learning tasks [119]. Here, we vary γ in $\{0.1, 0.2, 0.3, 0.4, 0.5\}$ to compare the performance of UDQN. The learning curves in terms of both mean (line) and standard deviation (shadow) of the results are shown in Figure 3.2, where each epoch corresponds to 20k Q-learning updates in Algorithm 3.2. The results on both tasks demonstrate two main points. First, UDQN with $\gamma = 0.2$ achieves the best performance. Second, UDQN with a lower γ learns faster than a higher one. This implies that, for our task, an agent that favors immediate rewards will achieve better performance. The possible reason is that our multi-MDP task is distinctly different from traditional tasks, as we discussed in Section 3.3.2. Moreover, unlike most of traditional tasks, the rewards in our task are not sparse (at each time step, the agent will receive a rating as reward).

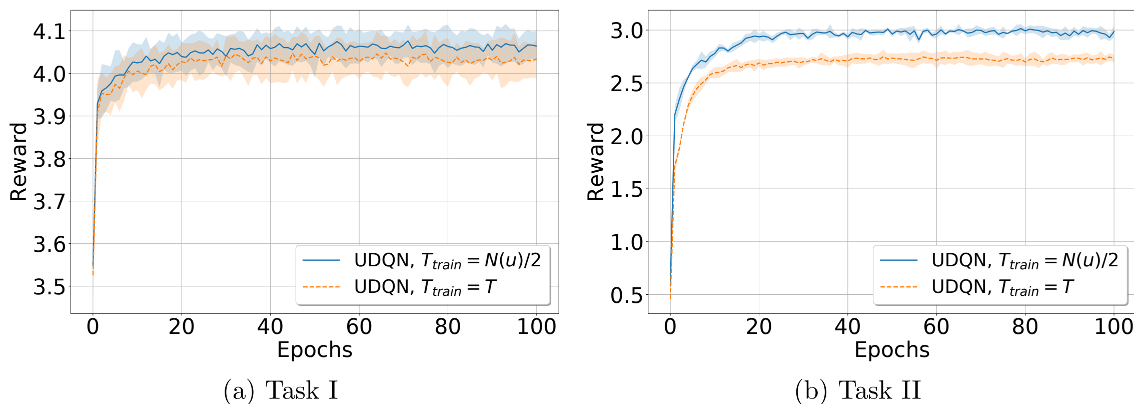
Effect of the Experience Replay

The training trick of experience replay was first proposed by [72], and has shown appealing performance improvements in some reinforcement learning tasks such as playing Atari games [90]. Here, we validate whether and how much it will benefit UDQN for totally different recommendation tasks. To do this, we vary the memory size $|\mathcal{M}| \in \{1, 10, 100, 1000\}$ to check how the performance changes. Note that



(a) Task I (b) Task II

Figure 3.3: Effect of the experience replay on the performance of UDQN.



(a) Task I (b) Task II

Figure 3.4: Effect of the parameter T_{train} on the performance of UDQN.

$|\mathcal{M}| = 1$ means that UDQN actually does not uses the trick of experience replay. The comparison results on both tasks are shown in Figure 3.3, which clearly tell us the same trend: $|\mathcal{M}| = 1$ achieves the best performance, and the performance will be consistently reduced when the memory becomes larger. The implies that, different from traditional tasks, the use of experience replay will significantly reduce the agent's performance for our recommendation tasks.

Effect of the Parameter T_{train}

Recall that T is the number of time steps in testing phase. Naturally, the parameter T_{train} is supposed to be the same number as T and the trained agent with this

setting is assumed to have better performance than others. Nevertheless, we find that a different setting that sets T_{train} distinctly w.r.t. different training users, may be a better choice. Here, we compare the performance of UDQN with two settings: $T_{train} = T$, and the default setting $T_{train} = N(u)/2$, where $N(u)$ denotes the number of observed ratings of user u in the dataset. As shown in Figure 3.4, UDQN with $T_{train} = N(u)/2$ performs better than $T_{train} = T$. This is mainly because that the agent with setting $T_{train} = N(u)/2$ can learn from more state transitions with rich diversity, and thus has stronger generalization ability.

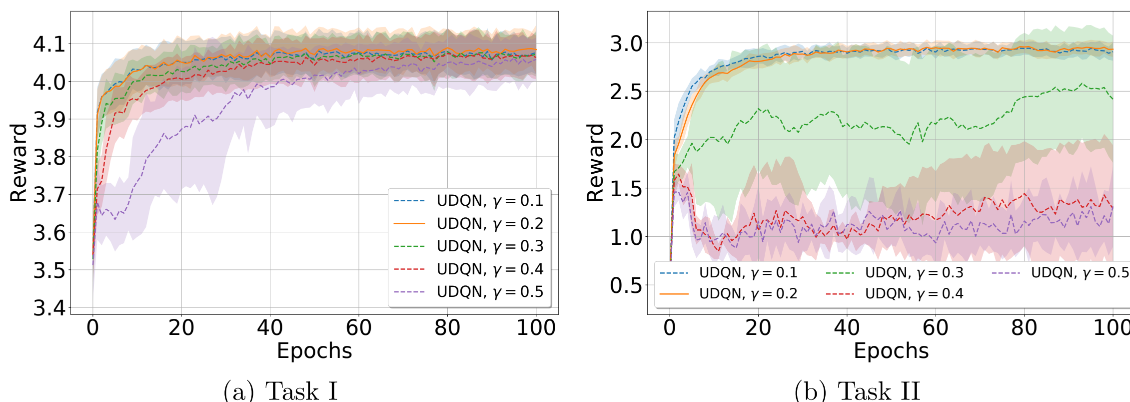
3.4.4 The Results of Cross-validation

In our original experimental settings, the dataset is divided into a training set and a testing set randomly for five times, which could result in overlapping in the testing set between different splits. To address this issue, we conduct an additional experiment using the 10-fold cross-validation to further compare the performance of different methods or of different hyper-parameter settings. We use the ML100K dataset as an example to conduct this experiment. More specifically, we randomly divide the candidate user set into 10 parts with equal size, where each part contains 10% candidate users. In each fold, we select one part as the testing set, and the remaining as the training set. The results are averaged over the 10 folds of data.

The results of different methods are shown in Table 3.4, which demonstrate very similar trends with the results under random split setting (Tables 3.2 and 3.3), although the absolute values are different. Importantly, the proposed method BUDQN still outperforms the baselines remarkably in the cross-validation setting. The results of different discount factor γ are shown in Figure 3.5, which demonstrate the same trend with the results under random split setting (Figure 3.2). With these results, we may expect that the results on other datasets or of other hyper-parameters under cross-validation setting will also show similar trends with the results under random

Table 3.4: The results of cross-validation on ML100K dataset.

Method	Task I		Task II	
	T=30	T=50	T=30	T=50
Random	3.522±0.094	3.533±0.084	0.447±0.039	0.436±0.038
Popular	3.844±0.052	3.832±0.053	2.543±0.099	2.361±0.097
SVD++	4.118±0.066*	4.048±0.069*	1.552±0.114	1.509±0.111
ICF	4.112±0.050	4.045±0.056	2.139±0.146	2.102±0.105
LinUCB	4.108±0.063	4.024±0.064	2.643±0.094*	2.382±0.084
DQN	4.072±0.070	4.006±0.068	2.630±0.098	2.459±0.090*
BUDQN	**4.214±0.062**	**4.125±0.045**	**3.219±0.097**	**3.016±0.091**
Improve	2.33%	1.90%	21.79%	22.65%



(a) Task I (b) Task II

Figure 3.5: Effect of the discount factor γ on the performance of UDQN in terms of cross-validation.

split setting.

3.5 Experiments on Implicit Feedback Recommendation Tasks

In this section, we empirically validate the efficacy of UDQN for the implicit feedback interactive top-k recommendation tasks, by conducting extensive experiments on real-world datasets.

Table 3.5: The statistics of datasets

Statistics	ML100K	ML1M	YMusic	WikiVote
#users	943	6,040	5,050	1,131
#items	1,682	3,952	1,000	2,330
#positive feedbacks	100,000	1,000,209	174,497	85,473

3.5.1 Experimental Setup

Datasets

We use four real-world benchmark datasets for conducting experiments, including two MovieLens datasets: ML100K and ML1M[3], one Yahoo! music dataset: YMusic[4], and one Wikipedia social network dataset: WikiVote[5]. The three datasets ML100K, ML1M and YMusic contain explicit ratings of items given by users. To perform implicit-feedback top-k recommendations, we convert the ratings to 1 to represent the positive feedbacks. The WikiVote dataset contains a directed voting network, where each edge denotes a vote relation. We consider the user-user vote pair (A, B) as a user-item feedback pair (user A, item B). For YMusic and WikiVote, we remove the users who have fewer than 20 positive feedbacks, so as to be consistent with the MovieLens datasets, as well as to ensure there is enough data for training and testing. To summarize, we present the statistics of all datasets in Table 3.5.

Evaluation Methodology

Without special mention, we set $T = 20$ and $k = 5$ for our evaluation, as such settings are usually appropriate for real-world recommendation scenarios on both

[3]http://grouplens.org/datasets/movielens/

[4]http://webscope.sandbox.yahoo.com/catalog.php?datatype=r

[5]http://snap.stanford.edu/data/wiki-Vote.html

desktop and mobile devices. The evaluation metric we used is Precision:

$$\text{Precision} = \frac{1}{T}\sum_{t=1}^{T}\frac{P_t}{k}, \tag{3.13}$$

where P_t denotes the number of positive feedbacks of the k recommended items at time step t. The Precision metric measures the accuracy of the top-k recommended items. Since k is small, we do not further measure the relative orders among the k items by using some other ranking metrics such as NDCG. In addition, we evaluate the interactive recommendation algorithms based on two different settings: cold-start and warm-start, which are described below.

In the cold-start setting, the agent has no information about target user's preferences at time step $t = 0$. To conduct this set of experiments, we split each dataset by randomly choosing 10% users as testing set $\mathcal{U}_{test}$, and the remaining as training set $\mathcal{U}_{train}$. The data of $\mathcal{U}_{train}$ is first used to pre-train the feature matrices U and V, then used to train the RL agent. However, in each episode, the initial state s_0 is set to zeros rather than the pre-trained U_u, so as to fit the cold-start setting.

In the warm-start setting, the agent already observes some feedbacks of target user at time step $t = 0$. To conduct this set of experiments, we split each dataset according to the following procedure. We first select the users who have at least 50 positive feedbacks as candidate set $\mathcal{U}_{can}$. We randomly choose 10% users from $\mathcal{U}_{can}$ as testing set $\mathcal{U}_{test}$, and the remaining as training set $\mathcal{U}_{train} = \mathcal{U}_{can} \backslash \mathcal{U}_{test}$. Also, we use $\mathcal{U}_{pre} = \mathcal{U} \backslash \mathcal{U}_{can}$ to denote the pre-training set. Then, all data of $\mathcal{U}_{pre}$ and 30 randomly chosen positive feedbacks of each user in $\mathcal{U}_{can}$ are used together to pre-train the feature matrices, so as to fit the warm-start setting. The remaining data of $\mathcal{U}_{train}$ and $\mathcal{U}_{test}$ is used for training and testing the RL agent, respectively.

For both recommendation settings, we repeat the data split procedure 5 times with different random seeds. We conduct each experiment based on the obtained 5

data splits, and calculate the mean and standard deviation of the results for evaluation.

Compared Methods

We derive four UDQN methods for validations and comparisons:

- **UDQN-MF**. The original UDQN that utilizes MF [59] to construct latent states.
- **UDQN-BPR**. A UDQN variant that utilizes BPR [97] to construct latent states.
- **UDQN-APPL**. A UDQN variant that utilizes APPL [65] to construct latent states.
- **UDQN-Concat**.A UDQN variant that utilizes the concatenation of both MF-based and APPL-based latent states.

We compare the UDQN methods with a variety of baselines:

- **Random**. A simple method that picks items randomly.
- **MF** [59]. A standard matrix factorization model that learns the scoring function $\hat{R}_{ui} = U_u^{\mathrm{T}} V_i$ by minimizing a pointwise squared error loss and picks items with largest scores.
- **BPR** [97]. A pairwise collaborative learning-to-rank method that learns $\hat{R}_{ui}$ by minimizing a pairwise ranking loss.
- **APPL** [65]. A joint collaborative learning-to-rank method that learns $\hat{R}_{ui}$ by minimizing pointwise and pairwise losses alternately.

- **LinUCB** [69]. A contextual bandit algorithm for news recommendation, which is extended to our problem by using the same learning framework of UDQN. Moreover, we concatenate U_u with V_i as its context vector.

- **DQN** [90]. A state-of-the-art deep reinforcement learning algorithm, which estimates Q-values based on the raw states.

Parameter Settings

We set the hyperparameters of compared methods based on the cross-validation on ML100K dataset in cold-start recommendation setting. For MF, BPR and APPL, we set the dimensionality of latent space $d = 64$, the regularization parameter $\lambda = 0.01$, and the learning rate $\alpha = 0.01$ (for updating feature vectors). For LinUCB, we set the parameter $\alpha = 0.5$. For DQN and UDQN, we set the discount factor $\gamma = 0.2$, the ϵ-greedy parameter $\epsilon = 0.1$, the learning rate $\alpha = 0.0001$ (for updating Q-networks), the number of time steps in training phase $T_{train} = 3T$, and use the same Q-network architecture (excluding the input layer), which consists of two hidden layers of 256 units with ReLU activation and an output layer of n units.

3.5.2 Performance Comparison

We first compare the performance of the proposed UDQN methods against baselines. The mean and standard deviation of the results on all datasets in both cold-start and warm-start recommendation settings are reported in Table 3.6. In each case, the bold font indicates the best performed method (UDQN-Concat in all cases), and the mark "*" denotes the best performed baseline (DQN in all cases). The reported p-value is computed by conducting the paired t-test with respect to these two methods. In addition, the relative improvement between them is shown in the last row. From the results in Table 3.6, we can summarize two main findings.

Table 3.6: Performance comparison on implicit-feedback top-k recommendation task.

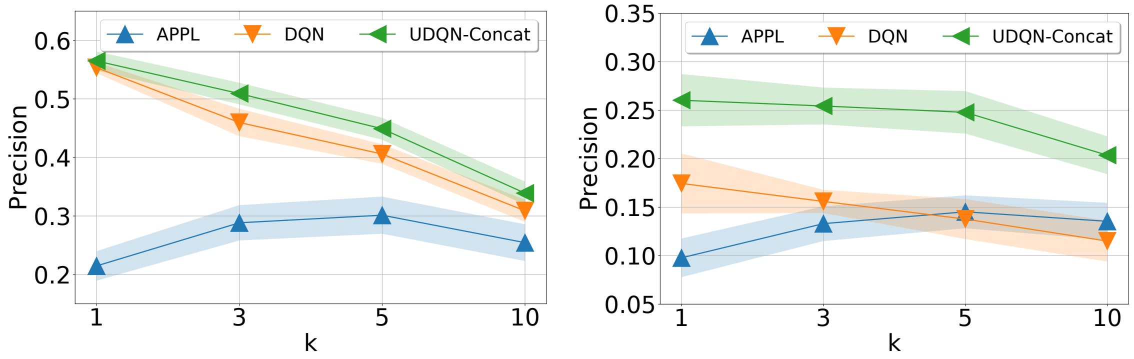
Methods	Cold-start Recommendation				Warm-start Recommendation			
	ML100K	**ML1M**	**YMusic**	**WikiVote**	**ML100K**	**ML1M**	**YMusic**	**WikiVote**
Random	0.067±0.006	0.043±0.001	0.035±0.002	0.031±0.006	0.078±0.010	0.048±0.001	0.062±0.004	0.043±0.002
MF	0.270±0.010	0.202±0.005	0.092±0.004	0.124±0.009	0.373±0.017	0.327±0.002	0.170±0.010	0.164±0.005
BPR	0.192±0.009	0.200±0.007	0.091±0.002	0.118±0.015	0.361±0.012	0.331±0.005	0.168±0.008	0.142±0.008
APPL	0.301±0.016	0.198±0.003	0.085±0.001	0.145±0.008	0.381±0.018	0.342±0.006	0.165±0.009	0.171±0.006
LinUCB	0.318±0.008	0.297±0.001	0.149±0.001	0.122±0.007	0.310±0.015	0.328±0.005	0.174±0.009	0.082±0.005
DQN	0.401±0.010*	0.354±0.009*	0.153±0.001*	0.163±0.007*	0.446±0.022*	0.356±0.015*	0.182±0.010*	0.173±0.015*
UDQN-MF	0.433±0.009	0.418±0.001	0.181±0.001	0.221±0.010	0.465±0.022	0.431±0.001	0.199±0.010	0.254±0.008
UDQN-BPR	0.420±0.008	0.402±0.001	0.175±0.002	0.210±0.008	0.458±0.018	0.435±0.002	0.202±0.009	0.223±0.008
UDQN-APPL	0.443±0.010	0.423±0.001	0.181±0.001	0.238±0.010	0.480±0.020	0.441±0.002	0.202±0.008	0.262±0.006
UDQN-Concat	**0.450±0.008**	**0.435±0.002**	**0.185±0.003**	**0.248±0.011**	**0.489±0.020**	**0.455±0.004**	**0.209±0.008**	**0.269±0.006**
p-value	3e-6	1e-7	9e-7	4e-6	5e-6	1e-5	7e-5	1e-6
Improve	11.97%	22.44%	21.40%	51.84%	9.69%	28.02%	14.94%	54.93%

Firstly, we find that all the proposed UDQN variants outperform the baselines remarkably and consistently in all cases. In particular, UDQN-Concat shows at least 9.69% (in the case of warm-start recommendation on ML100K dataset) and up to 54.93% (in the case of warm-start recommendation on WikiVote dataset) improvement over the best performed baseline. This finding strongly highlights the efficacy of our approach to interactive top-k recommendations.

Secondly, we find that the performance of UDQN-MF, UDQN-BPR and UDQN-APPL is basically consistent with the three collaborative filtering models: MF, BPR and APPL. For instance, in the case of ML100K dataset and cold-start setting, the performance of the three UDQN variants is ranked as: UDQN-APPL > UDQN-MF > UDQN-BPR, same as the ranking of the three collaborative filtering models: APPL > MF > BPR. This finding implies that the performance of UDQN is partially dependent on the quality of the user-specific latent states learned with the collaborative filtering models. It also supports our claim made previously that the performance of UDQN will be improved consistently as long as the adopted collaborative filtering model can better capture users' preferences.

3.5.3 Parameter Analysis

Recall that in previous experiments, we set $T = 20$ and $k = 5$ in default for the evaluation on T-step interactive top-k recommendations. In this subsection, we validate how UDQN performs with different settings of T and k. We use UDQN-Concat as a representative to conduct this part of analysis. Also, we choose APPL from non-RL baselines and DQN as representatives for the comparison. This set of experiments are conducted on ML100K and WikiVote datasets in cold-start recommendation setting.



(a) ML100K (b) WikiVote

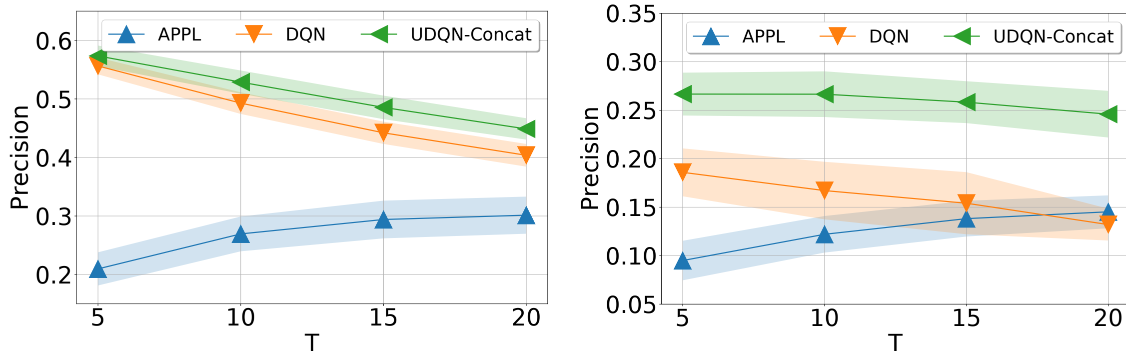
Figure 3.6: Performance of UDQN-Concat for different k.

Performance for Different k

We first vary $k \in \{1, 3, 5, 10\}$ to evaluate the performance of the compared methods, while fixing all the other experimental settings as default. The results are shown in Figure 3.6, where the shade indicates the standard deviation over 5 runs. We can see that UDQN-Concat outperforms other two methods consistently. Also, the Precision of top-k items will decrease when k becomes larger, which is reasonable in top-k recommendations. Besides, the non-RL method APPL performs relatively worse with a small k due to cold-start issue, since there is no sufficient feedback data for training the prediction model during the interactive process (only kT feedbacks will be collected).

Performance for Different T

We now vary $T \in \{5, 10, 15, 20\}$ to evaluate the performance of the compared methods, while fixing all the other experimental settings as default. The results are shown in Figure 3.7. Again, UDQN-Concat outperforms other two methods consistently, and the relative improvement over DQN is continuously increasing when T becomes larger. This indicates that our approach is able to model future wards more effectively, even with the increasing difficulty of the T-step leaning task. Besides, APPL



(a) ML100K (b) WikiVote

Figure 3.7: Performance of UDQN-Concat for different T.

shows different trend due to cold-start issue, similar to the trend with respect to k.

3.6 Conclusions and Discussions

In this work, we study a T-step interactive recommendation problem for general recommender systems. Different from the existing works, we propose a novel user-specific deep RL approach to the problem, which effectively models the user-specific information of both personalized preferences and collaborative relationships. More specifically, we develop two user-specific deep Q-learning methods, UDQN and BUDQN, to learn recommendation policies based on the user-specific latent states constructed by matrix factorization (MF). We conduct extensive experiments to evaluate our approach on three real-world datasets in terms of both explicit-feedback and implicit-feedback recommendation tasks. The comprehensive results demonstrate that it achieves remarkable improvements over a wide variety of state-of-the-art recommendation methods.

Although MF is well studied in traditional recommendation domains, it is based on the paradigm of supervised learning and has not been investigated in the context of RL. Working well in supervised learning tasks (e.g., rating prediction task) does not means it can also work in RL tasks (e.g., our interactive recommendation task). Our

work provides a positive answer to this research question by successfully integrating MF with RL. Moreover, our approach is generic, which can be deemed as a general framework for RL-based recommendations. In stead of DQN, it is easy to employ other state-of-the-art RL algorithms to estimate the optimal policy based on the proposed MF-based latent states and actions. On the other hand, in stead of the standard MF model, it is easy to employ other more complex latent factor models to construct the latent states and actions.

Chapter 4

Graph Convolutional Q-network: Modeling Personalization and Collaboration via GCN-based Representation Module

In this work, we propose an alternative way to model personalization and collaboration by building graph-structured representations of states and actions according the user-item bipartite graph. We develop an effective end-to-end agent, termed Graph Convolutional Q-network (GCQN), which is able to directly approximate the optimal action-value function based on the input of graph-structured representations. In particular, GCQN successfully leverages a variant of graph convolutional network (GCN) to transform the low-level graph-structured representations to higher-level vector representations of states and actions. We show that GCQN achieves significant improvements over the existing methods, across different datasets and task settings, with acceptable computation cost.

4.1 Introduction

Reinforcement learning (RL) [119] is a promising approach to recommender systems, which aims to build a recommendation agent that is able to adaptively recommend

potentially interesting items to users in a sequential manner. Compared to non-RL recommendation engines, a key nature of RL agents is that they are able to not only capture users' dynamic preferences via continuous user-agent interactions, but also learn farsighted policies that achieve maximal long-term rewards from users. Recently, researchers introduced the techniques of deep RL (e.g., deep Q-networks [90]) to design novel recommendation agents with good generalization ability and scalability. These modern RL agents, which use neural networks to approximate the optimal action-value functions or policies, have shown great potential in a variety of recommendation domains ranging from news feeds to E-commerce sites [160, 158, 19, 20].

Despite their successes, however, there is an important problem that has been rarely noticed and investigated in these prior works. That is, *how to effectively represent the states and actions for an RL recommendation agent, according to specific characteristics in recommender systems, e.g., user-item bipartite graph?* This problem is very crucial to the learning of effective recommendation policies. As pointed out in [119], the performance of an RL system is highly dependent on the state/action representations adopted in the system. Generally speaking, good representations should capture sufficient and useful task-related information, in order to facilitate the agent to complete the specific decision making task of interest. In fact, the impressive performance of deep RL agents in many game-based tasks, such as Atari [90, 144] and Go [110], are largely because that they successfully learn useful high-level state representations from raw image inputs by using deep convolutional neural networks (CNNs).

The above observations motivate us to revisit the existing works in RL-based recommender systems, and analyze the different types of state/action representations used by them. In some works [160, 19], the states (actions) are represented by some handcrafted features of users (items). Although these handcrafted representations

have good interpretability, two obvious weaknesses limit their applicability: (1) they require a large amount of human effort for feature engineering; and (2) they are only appropriate to feature-rich scenarios such as news recommendations. On the other hand, more researchers [158, 141] leverage neural networks to learn high-level vector representations from the raw states (e.g., item sequences) and actions. Such neural-network-based representations are learned in an end-to-end fashion, which only require item ids for inputs and hence can be applied to most recommendation domains. Unfortunately, a common weakness of the existing RL-based methods is that *they represent the states/actions of each individual user in isolation, such that the collaborative relationships (e.g., similarities) between different users are not explored and exploited*. This weakness significantly limits the performance of existing methods in learning personalized recommendation policies for the entire user community, as shown in our experiments.

In this work, we propose a novel idea to overcome the weakness of the existing RL-based recommendation methods. We suggest to design effective graph-structured representations for states and actions based on user-item bipartite graph, from which personalized recommendation policies can be learned. We start by following the existing work [120] to define the raw state as an item sequence observed currently, which indicates that our approach is applicable to most recommendation domains. We then build a graph-structured state based on the raw item sequence by replacing each item i in the sequence with a specific sub-graph $G(i)$ that consists of item i and its neighborhood in the whole user-item bipartite graph G. Figure 4.1 illustrates how to build a graph-structured state for target user u through a toy example. *Such graph-structured state has two notable qualities: (1) it captures the rich structural information in user-item bipartite graph, which enables the agent to discover collaborative preferences from target user u's neighbors (e.g.,* u_1, u_2, u_3*); and (2) it preserves the sequential information in original item sequence, which enables the*

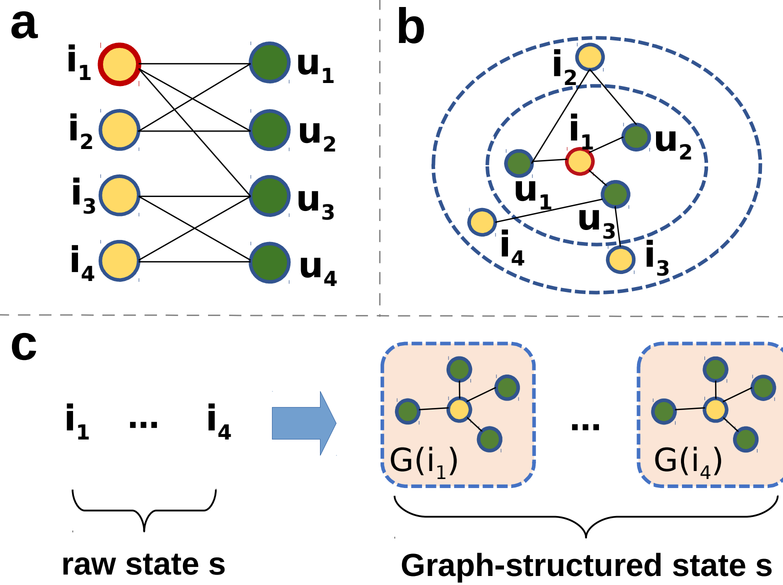


Figure 4.1: A toy example to illustrate how to build a graph-structured state for target user u. (a) The whole user-item bipartite graph G. (b) The sub-graph $G(i_1)$ that consists of item i_1 and its neighborhood in G. (c) Building a graph-structured state $s = \{G(i_1), ..., G(i_4)\}$ based on the raw state $s = \{i_1, ..., i_4\}$.

agent to model the dynamic preferences of target user u. In the same way, each action can be represented by a specific sub-graph instead of a single item.

To effectively learn policies based on the graph-structured representations, we develop an end-to-end RL agent, termed Graph Convolutional Q-network (GCQN). GCQN is able to transform the graph-structured states and actions to high-level vector representations and finally predict the action-values based on them (see Figure 4.2). More specifically, we propose a variant of graph convolutional network (GCN) by incorporating the ideas of both GraphSage [38] and GAT [128]. We use this GCN module to exploit the structural information in the graphs of states and actions, and output some graph-aware vector representations. We then take advantage of the gated recurrent unit (GRU) [22] with a self-attention mechanism [73], to process the sequence of graph-aware representations and produce a higher-level vector representation of the state. Finally, the concatenation of the high-level state representation

and action representation is fed into a multilayer perceptron (MLP) to predict the action-value of the state-action pair.

To summarize, we offer the following main contributions:

- We provide a new way to improve the existing RL based recommender systems by building graphs to represent states and actions. The proposed graph-structured representations capture both structural information and sequential information, such that the factors of collaboration and personalization are effectively embedded. As a result, the graph-structured representations offer much bigger opportunity for the agents to estimate optimal policies, in contrast to the existing unstructured representations.

- We develop an effective end-to-end agent, GCQN, which is able to approximate the optimal action-value function based on the inputs of graph-structured representations. GCQN successfully exploits the structural and sequential information by leveraging GCN and GRU, respectively.

- We conduct extensive experiments on three real-world datasets to validate our approach. The experimental results demonstrate that: (1) graph-structured representations help a lot in learning farsighted recommendation policies; and (2) GCQN is effective and robust, which achieves significant performance gain over state-of-the-art baselines across different datasets and task settings.

4.2 Preliminaries

We formulate the RL based recommendation problem as follows. Suppose we have a recommender system with a set of m users $\mathcal{U} = \{1, ..., m\}$, a set of n items $\mathcal{I} = \{1, ..., n\}$, and an observed user-item implicit feedback matrix $Y \in \mathbb{R}^{m \times n}$, where $y_{ui} = 1$ if user u gives a positive feedback on item i (clicking, watching, etc.), and

$y_{ui} = 0$ otherwise. We consider an episodic RL task [119], where in each episode, a recommendation agent interacts with a target user u at discrete time steps $t = 0, ..., T-1$. At each time step t, the agent observes a state s_t (the current situation the agent faces), and accordingly takes an action a_t (i.e., recommends an item $a_t \in \mathcal{I}$) based on its policy π (indicating how to choose actions given states). One step later, as a consequence of its action a_t, the agent receives a reward $r_{t+1} = y_{ua_t}$ from user u and observes next state s_{t+1}.

Given the data of training users $\mathcal{U}_{train} \subset \mathcal{U}$, our goal is to learn an optimal policy that is able to maximize the cumulative rewards received in a T-step episode, $\sum_{t=0}^{T-1} r_{t+1}$, for testing users $\mathcal{U}_{test} = \mathcal{U} \backslash \mathcal{U}_{train}$. To estimate the policy, in this work we follow a well-adopted approach that uses a neural network $Q(s, a; \theta)$ (i.e., Q-network) to approximate the optimal action-value function $Q^*(s, a)$ (corresponding to an optimal policy π^*) [72, 119, 89, 90] via Q-learning [145].

4.3 Graph Convolutional Q-network

In this section, we start by formally defining the graph-structured states and actions. We then present the GCQN model. Lastly, we show the training algorithm of GCQN.

4.3.1 Representing States and Actions as Graphs

We first define the raw state s_t of target user u as an item sequence, $\{a_0, ..., a_{t-1}\}$, denoting the items that user u has consumed before time step t[1], and define the raw action a as a unique item. We then build graph-structured representations by transforming the raw state $\{a_0, ..., a_{t-1}\}$ to a sequence of sub-graphs $\{G(a_0), ..., G(a_{t-1})\}$, and by transforming the raw action a to a sub-graph $G(a)$. Here, each sub-graph

[1]Intuitively, a more realistic state might consider the corresponding feedback sequence $\{y_{ua_0}, ..., y_{ua_{t-1}}\}$ simultaneously. However, how to effectively exploit such feedback information to estimate action-values is a big challenge. Simple use (e.g., DEERS [158]) will heavily degenerate the agent's performance in our tasks. We leave the exploration to this problem in the future work.

$G(i)$ consists of item i and its neighborhood in the whole user-item bipartite graph G (as shown in Figure 4.1)[2].

It is worth noting that the proposed graph-structured representations have several good qualities. First of all, after taking action a_t in the graph-structured state $s_t = \{G(a_0), ..., G(a_{t-1})\}$, the agent will observe next graph-structured state $s_{t+1} = \{G(a_0), ..., G(a_{t-1}), G(a_t)\}$ at time step $t + 1$. This implies that the state transition actually meets the Markov property of a Markov decision process (MDP), which provides well guarantees for applying Q-learning based methods to solve the RL based recommendation problem [119]. Secondly, the graph-structured state also inherits the sequential property of the original item sequence, which implies that sequential models such as RNNs can be employed to capture the dynamic and evolving preferences of target user. Last but not least, the graph-structured states and actions capture the rich structural information in the user-item graph, which enables the agent to collaboratively explore personalized recommendation policies for diverse users.

4.3.2 The GCQN Model

We now describe the GCQN model, an end-to-end RL agent that takes the graph-structured representations of state-action pair (s_t, a) as input, and outputs the predicted action-value $Q(s_t, a)$. The network architecture of GCQN is illustrated in Figure 4.2. We will elaborate on GCQN in the following.

Embedding Layer. We first map all users and items to a low-dimensional vector space. Each user u and each item i is described by a unique embedding vector $\mathbf{e}_u \in \mathbb{R}^d$ and $\mathbf{e}_i \in \mathbb{R}^d$, respectively. The embeddings of users and items are implemented as two simple lookup tables: $\mathbf{E}_u = [\mathbf{e}_1, ..., \mathbf{e}_m]$ and $\mathbf{E}_i = [\mathbf{e}_1, ..., \mathbf{e}_n]$. These embeddings

[2]In practice, we only use training users' data to construct graph G, in order to ensure no test information is used during training.

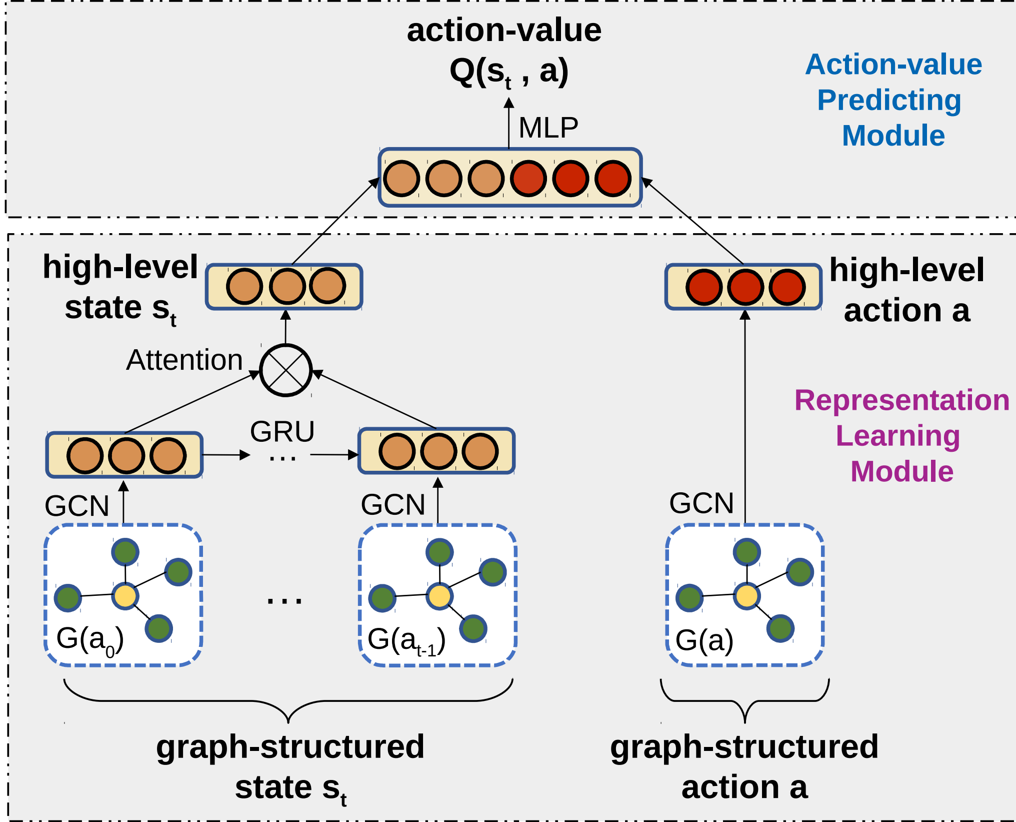


Figure 4.2: Overview of the GCQN model.

are treated as the raw features of nodes in the graphs, which are randomly initialized and trained in an end-to-end fashion.

GCN Layer. We develop an variant of GCN by taking advantage of the ideas of both GraphSage [38] and GAT [128]. The goal of this GCN module is to process each sub-graph $G(i)$, and produce a graph-aware representation of item i, $\mathbf{x}_i \in \mathbb{R}^d$:

$$\mathbf{x}_i \leftarrow \text{relu}(\mathbf{W}_{fc}[\mathbf{e}_i \oplus \mathbf{e}_{\mathcal{N}(i)}] + \mathbf{b}_{fc}), \tag{4.1}$$

where $\mathbf{W}_{fc} \in \mathbb{R}^{d\times 2d}$ and $\mathbf{b}_{fc} \in \mathbb{R}^d$ are the trainable weights and biases of a fully connected (FC) layer, $\mathbf{e}_i$ is the embedding of item i, $\oplus$ denotes the concatenation operation, and $\mathbf{e}_{\mathcal{N}(i)} \in \mathbb{R}^d$ is the neighborhood vector that is computed by an attention-based aggregator AGG^{att}:

$$\mathbf{e}_{\mathcal{N}(i)} \leftarrow \sum\nolimits_{w\in\mathcal{N}(i)} \alpha_{iw}\mathbf{e}_w, \tag{4.2}$$

where $\mathcal{N}(i)$ denotes the set of the 1-hop neighbors of item i in $G(i)$, $\mathbf{e}_w$ is the embedding of user w, and α_{iw} is the attention score that determines how much feature information of user w will be passed to item i. We adopt the Concat attention mechanism [79] to compute the attention score α_{iw}:

$$\alpha_{iw} \leftarrow \frac{\exp\left(\mathbf{w}_a^\top \tanh(\mathbf{W}_a[\mathbf{e}_i \oplus \mathbf{e}_w])\right)}{\sum_{v \in \mathcal{N}(i)} \exp\left(\mathbf{w}_a^\top \tanh(\mathbf{W}_a[\mathbf{e}_i \oplus \mathbf{e}_v])\right)}, \tag{4.3}$$

where $\mathbf{W}_a \in \mathbb{R}^{d\times 2d}$ and $\mathbf{w}_a \in \mathbb{R}^d$ are trainable weights for attention, and $\cdot^\top$ is the transpose operation.

In a real-word recommender system, the size of $\mathcal{N}(i)$ may vary dramatically over different items i (e.g., follow the long-tail distribution). To make the computation more efficient, as many existing works did [38], we uniformly sample a fixed-size set of user neighbors for each item i, instead of using its full neighborhood. That is, we redefine $\mathcal{N}(i)$ as a fixed L-size, uniform sample from the full set $\mathcal{N}^+(i) = \{u : y_{ui} = 1\}$. Note that $\mathcal{N}(i)$ is formed with different uniform samples at each iteration during training, and will contain duplicates when $L > |\mathcal{N}^+(i)|$.

In our experiments, we also examined the mean-based aggregator $\mathrm{AGG}^{\mathrm{mean}}$ and the pooling-based aggregator $\mathrm{AGG}^{\mathrm{pool}}$ proposed in [38]. However, these two aggregators did not show comparable performance against our attention-based one (see Table 4.3 for comparison), as they failed to model the different influence strength from each user $v \in \mathcal{N}(i)$ to item i. For example, although both user A and user B give the same positive feedback on item i, they might have distinctly different embedding vectors that should be treated distinguishingly when aggregating features.

By applying the GCN module to the action graph $G(a)$ and the state graphs $\{G(a_0), ..., G(a_{t-1})\}$, we obtain a graph-aware vector representation $\mathbf{x}_a$ of action a, and a sequence of graph-aware vector representations $\mathbf{x}(s_t) = \{\mathbf{x}_{a_0}, ..., \mathbf{x}_{a_{t-1}}\}$ of state s_t, respectively. Both will be fed to the next layers.

GRU Layer. To model the sequential information in the state, as the exiting work did [158], we leverage a gated recurrent unit (GRU)[3] to further process the graph-aware state representation $\mathbf{x}(s_t) = \{\mathbf{x}_{a_0}, ..., \mathbf{x}_{a_{t-1}}\}$, which can be simply denoted by $\{\mathbf{x}_0, ..., \mathbf{x}_{t-1}\}$ for notational convenience. The goal of this GRU module is to transform $\{\mathbf{x}_0, ..., \mathbf{x}_{t-1}\}$ to a sequence of hidden vectors $\{\mathbf{h}_0, ..., \mathbf{h}_{t-1}\}$, where $\mathbf{h}_j \in \mathbb{R}^d$ is abstractly computed as:

$$\mathbf{h}_j \leftarrow \text{GRU}(\mathbf{h}_{j-1}, \mathbf{x}_j). \tag{4.4}$$

Furthermore, we employ a self-attention mechanism [73] to capture the importance of different items in the state, and summarize a final state representation $\mathbf{h}_{s_t}$:

$$\mathbf{h}_{s_t} \leftarrow \sum_{j=0}^{t-1} \beta_j \mathbf{h}_j, \tag{4.5}$$

where β_j is the attention score that indicates how much feature information of item a_j will be extracted, which is computed by:

$$\beta_j \leftarrow \frac{\exp\left(\mathbf{w}_{sa}^\top \tanh(\mathbf{W}_{sa}\mathbf{h}_j)\right)}{\sum_{l=0}^{t-1} \exp\left(\mathbf{w}_{sa}^\top \tanh(\mathbf{W}_{sa}\mathbf{h}_l)\right)}, \tag{4.6}$$

where $\mathbf{W}_{sa} \in \mathbb{R}^{d \times d}$ and $\mathbf{w}_{sa} \in \mathbb{R}^d$ are trainable weights in the self-attention. With this attention mechanism, the GRU layer is able to autonomously select more important features from the input sequence, and thus help the agent capture the user's dynamic preference at each time step.

MLP Layers. Once the state vector representation $\mathbf{h}_{s_t} \in \mathbb{R}^d$ and the action vector representation $\mathbf{x}_a \in \mathbb{R}^d$ are ready, we employ a multilayer perceptron (MLP) (with architecture $2d \to \cdots \to 1$) to fuse useful feature information from both of them and

[3] We chose GRU instead of long-short term memory (LSTM) and simple RNN because GRU has shown advantages over them in many recommendation tasks [44, 158].

predict the final action-value $Q(s_t, a)$:

$$\mathbf{c}_1 \leftarrow \text{relu}(\mathbf{W}_1[\mathbf{h}_{s_t} \oplus \mathbf{x}_a] + \mathbf{b}_1),$$
$$\cdots$$
$$Q(s_t, a) \leftarrow \mathbf{w}_l^\top \mathbf{c}_{l-1} + \text{b}_l, \tag{4.7}$$

where $\mathbf{c}_i$, $\mathbf{W}_i$ (or $\mathbf{w}_i$) and $\mathbf{b}_i$ (or b_i) denote the outputs, trainable weights and biases of the i-th layer of MLP, respectively.

4.3.3 Training Algorithm

To train GCQN, i.e., the Q-network denoted by $Q(s, a; \theta)$, we adopt a Q-learning [145] based algorithm, which minimizes the following loss function:

$$\mathcal{L}(\theta) = \mathbb{E}_{s,a,r,s'} \left[(y - Q(s, a; \theta))^2 \right], \tag{4.8}$$

where θ denote all the trainable parameters of the Q-network, $y = r + \gamma \max_{a'} Q(s', a'; \theta^-)$ is the Q-learning target for current iteration, γ is the discount factor that balances the importance between future rewards and immediate rewards, and θ^- are the Q-network parameters from previous iteration, which are held fixed when performing optimization. In practice, instead of optimizing the full expectations in the loss function, a more convenient way is to perform stochastic gradient descent (SGD) on a sampled transition (s, a, r, s') [89, 90]:

$$\theta \leftarrow \theta + \alpha \left[y - Q(s, a; \theta) \right] \nabla_\theta Q(s, a; \theta). \tag{4.9}$$

We present the training algorithm of GCQN in Algorithm 4.1. To make the Q-network converge well, sufficient transitions that involve all possible states and actions are needed for Q-learning updates [119]. To this end, in each episode, we uniformly sample a user u from training set $\mathcal{U}_{train}$ as the current target user, which will interact with the agent and generate corresponding states and rewards. To

Algorithm 4.1: The Training Algorithm of GCQN

Input: training user set $\mathcal{U}_{train}$, feedback data Y
Output: trained Q-network Q
1 **for** *episode* $= 1, ..., N$ **do**
2 Uniformly sample a target user u from $\mathcal{U}_{train}$
3 Initialize state $s_0 = \{i_c\}$, where i_c is a random item
4 **for** $t = 0, ..., T-1$ **do**
5 Recommend the ϵ-greedy item a_t w.r.t. $Q(s_t, a; \theta)$
6 Set reward $r_{t+1} = y_{ua_t}$ and state $s_{t+1} = s_t \cup \{a_t\}$
7 Update Q's weights θ according to **Equation 4.9**

ensure exploration, in each state s_t, the agent uses a ϵ-greedy policy that selects a greedy action $a_t = \arg\max_a Q(s_t, a)$ with probability $1-\epsilon$ and a random action with probability ϵ. At the testing stage, the trained Q-network agent can be used to make real-time recommendations for any user $v \in \mathcal{U}_{test}$. It only needs to interact with user v, observe state s_t, predict the action-values $Q(s_t, a)$ for all actions, and recommend the greedy item $a_t = \arg\max_a Q(s_t, a)$, at each time step t.

Time Complexity. In the inner for-loop in Algorithm 4.1, the computation time is mainly taken in computing Q-values for all available items (line 5), and in updating Q-network (line 7). The cost of computing Q-values is $O(n|\theta|)$, where $|\theta|$ is the number of Q-network parameters and n is the number of items, and the cost of updating Q-network is $O(|\theta|)$. Therefore, the time complexity of training GCQN in worst case is $O(NTn|\theta|)$, where N is the number of episodes and T is the number of time steps in each episode.

4.4 Experiments

We conducted extensive experiments to validate the proposed GCQN method. In this section, we describe our experiments and show the results and analysis.

Table 4.1: The statistics of datasets.

Statistics	LastFM	ML1M	Pinterest
#users	1,874	6,040	13,397
#items	2,828	3,416	9,359
#observed feedbacks	71,411	999,611	494,523

4.4.1 Experimental Setup

Datasets. We employ three public recommendation datasets: **LastFM**[4] [12] (a music recommendation dataset collected from the Last.fm website), **ML1M**[5] [39] (a movie recommendation dataset collected from the MovieLens website), and **Pinterest**[6] [34, 42] (an image recommendation dataset collected from the Pinterest website). All datasets contain user-item feedback data. Since we focus on implicit-feedback recommendations, we follow the common practice to treat users' diverse behaviors (listening, rating, or pinning) on items (artists, movies, or images) as a unified implicit positive feedback[7]. To ensure there is enough data for training and testing reinforcement learning agents, we first remove the items with fewer than 5 feedbacks for all datasets, and then remove the users with fewer than 5, 20 and 30 feedbacks for LastFM, ML1M and Pinterest, respectively (in proportion to the number of items in each dataset). A summary of characteristics of the datasets is given in Table 4.1.

Evaluation Protocols. To evaluate reinforcement learning based recommendation algorithms, we assume the observed feedbacks in the datasets are unbiased, similar to [69, 159]. Since not all unobserved items are truly negative, we randomly select 1000 unobserved (u, i) pairs of user u as the negative feedbacks ($y_{ui} = 0$), similar to [26]. In each T-step episode of the user-agent interactions, the agent is forced to pick

[4]https://grouplens.org/datasets/hetrec-2011/

[5]https://grouplens.org/datasets/movielens/1m/

[6]https://github.com/hexiangnan/neural_collaborative_filtering/tree/master/Data

[7]The explicit feedbacks in LastFM (listening counts) and ML1M (ratings) are converted to 1.

items from the available item set that consists of the 1000 sampled negative items and the observed positive items. We conduct experiments for cold-start recommendation scenario, which implies that the agent has no feedback data of target user at time step $t = 0$, i.e., at the beginning of each T-step episode. We split each dataset by randomly choosing 80% users as the training set $\mathcal{U}_{train}$, and the remaining 20% users as the testing set $\mathcal{U}_{test}$. We conduct each experiment on 5 data splits obtained with different random seeds. The evaluation metric we used is the mean of the rewards received in a T-step episode, which is equivalent to the ratio of positive items in the recommended items during the episode. The final metric is obtained by taking its average over all testing users and 5 data splits.

Baselines. To comparatively evaluate our GCQN, we carefully choose a number of representative **RL-based methods** from the literature, which can be applied to our task with reasonable adaptions or extensions:

- **DEERS** [158]. This is a DQN-based method designed for product recommendation in E-commerce. It utilizes GRU to learn state representations from positive- and negative-feedback item sequences.

- **LSTM-Q**. This method learns the state representations from item sequences by using LSTM [45], and learns the action representations from item ids by using embedding layer.

- **GRU-Q**. This method is similar to LSTM-Q. The only difference is that it utilizes GRU [22] to learn state representations.

- **AttLSTM-Q**. This method extends LSTM-Q with a self-based attention mechanism [73].

- **AttGRU-Q**. This method extends GRU-Q with the same self-based attention mechanism.

All of the above baselines are Q-network agents. The major difference between them and our GCQN is that they only use RNNs to learn state representations and use a simple embedding layer to learn action representations. To apply them to our task, as well as for fair comparisons, we train these agents using the same Q-learning algorithm of GCQN (i.e., Algorithm 4.1). This enables us to focus on validating the efficacy of the proposed graph-structured representations.

We also compare several **non-RL methods** for reference:

- **SVD** [59] is a matrix factorization model that uses the inner product of latent feature vectors to predict user-item relevance scores. To fit our task, we first train the item feature vectors based on training data by using SVD and uniform negative sampling, which will be held fixed during testing. When performing T-step recommendations for a testing user v, the feature vector of user v is randomly initialized, and is always updated based on the fixed item feature vectors according to the new feedback data at each time step.

- **Popular** picks items with most positive feedbacks. It is a simple but strong baseline in many recommendation tasks.

- **Random** picks items randomly. It can be seen as an indicator that reveals the difficulty of the task itself.

It is worth noting that our RL-based recommendation problem is a fully cold-start online learning task. Thus, the majority of the existing supervised learning based methods are not able to provide meaningful recommendations for our task[8].

Parameter Settings. We implement the compared reinforcement learning based methods in PyTorch. Since all of them are trained with the same Q-learning algo-

[8]In fact, we have tried NCF [42] and NGCF [142] that utilize MLP and GCN to replace the inner product in SVD, respectively. However, these two complex models failed to produce better results than SVD in a reasonable amount of training time. Since all of them are supervised learning based methods, we only report the results of SVD here.

Table 4.2: The average reward received in T-step episodes ($T = 20$).

Type	Method	LastFM	ML1M	Pinterest
Non-RL	Random	0.036	0.123	0.034
	Popular	0.330	0.608	0.175
	SVD	0.151	0.285	0.084
RL	DEERS	0.243	0.511	0.095
	LSTM-Q	0.336	0.621	0.171
	GRU-Q	0.346	0.626	0.178
	AttLSTM-Q	0.354	0.632	0.181
	AttGRU-Q	0.375	0.633	0.191
Graph RL	GCQN	**0.404**	**0.658**	**0.215**

rithm, we use the LSTM-Q as a base model to tune some common hyperparameters using grid search, which will be fixed to all methods. This setting is reasonable as it enables us to make a fair comparison on the particular representation learning modules in different methods. More specifically, we adopt the Adam optimizer to update Q-networks. All parameters in Adam are set to the default values in PyTorch, with one exception of learning rate $\alpha = 0.0001$. Other shared hyperparameters are set as follows: the embedding size $d = 64$, the discount factor $\gamma = 0.5$, the ϵ-greedy parameter $\epsilon = 0.1$, the number of training episodes $N = 2 \times 10^5$, and the architecture of the MLP $128 \rightarrow 64 \rightarrow 32 \rightarrow 1$. For DEERS, we adopt the same architecture as in the original paper [158]. For GCQN, we set the neighborhood sample size $L = 10$. For SVD, we set the latent factor dimension as 64, the regularization parameter as 0.001, and the learning rate as 0.01.

4.4.2 Comparison Results

The comparison results of all methods, in terms of the average reward received in T-step episodes ($T = 20$), are reported in Table 4.2. The best performing method is highlighted in bold font. The proposed GCQN method shows significant margins

over the baselines on all datasets.

- Our GCQN shows remarkable advantages over the baselines on all datasets, including AttLSTM-Q and AttGRU-Q whose major differences between GCQN is that they do not use the graph-structured state/action representations. More specifically, the improvements of GCQN over the best performing baseline AttGRU-Q are about 7.7%, 3.9% and 12.5% on LastFM, ML1M and Pinterest datasets, respectively[9]. *This clearly demonstrates that the proposed graph representations of states and actions are very helpful to learning reinforcement learning based recommendation policies, by effectively exploiting them with GCN networks.*

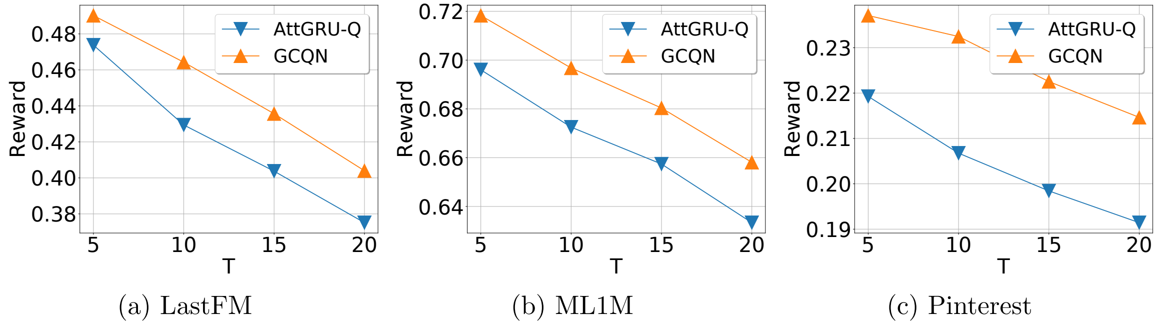
- AttLSTM-Q and AttGRU-Q show the second-class performance, with notable improvements over LSTM-Q and GRU-Q. This demonstrates the efficacy of using attention mechanisms to help the agent estimate action-values. AttGRU-Q (GRU-Q) performs better than AttLSTM-Q (LSTM-Q), which is consistent with many other recommendation tasks [44, 158].

- DEERS shows much worse performance than other reinforcement learning methods, which implies that it is not beneficial to divide the item sequence into two sub-sequences of positive and negative feedbacks. SVD shows poor performance as it suffers from cold-start, while Popular is a stronger baseline.

4.4.3 Model Analysis

Performance for Different T. We also show the performance of GCQN when varying $T \in \{5, 10, 15, 20\}$, in comparison to the best performing baseline AttGRU-Q. The results are shown in Figure 4.3, from which we observe that:

[9]Note that in the famous Netflix Prize, the winner only improves the baseline 10%!



(a) LastFM (b) ML1M (c) Pinterest

Figure 4.3: Comparison for different $T \in \{5, 10, 15, 20\}$. Our GCQN consistently shows remarkable improvements over AttGRU-Q.

- The rewards of both methods decrease significantly when T increases from 5 to 20. This is mainly because of two reasons: (1) the recommendation task of T-step episode becomes harder when T increases, since a more farsighted policy needs to be learned, and (2) the observed positive feedbacks in the dataset are limited, which reduces the chance of picking positive items especially at later time steps, because the agent cannot select repetitive items to recommend in our settings.

- Our GCQN method consistently shows significant improvements over AttGRU-Q for different T. *This verifies again the effectiveness and robustness of GCQN, across different datasets and task settings.*

Running Time Comparison. We now compare the running time of all reinforcement learning based methods on a single-GPU machine. Since all of them are Q-networks using same Q-learning algorithm, their time complexity is $O(NTn|\theta|)$ (as analyzed previously). The only difference between these methods is the number of Q-network parameters $|\theta|$. Figure 4.4 shows the average training time of the methods on each dataset (for $T = 20$). Overall, the training cost of GCQN is close to DEERS, nearly 1.5 times higher than AttLSTM and AttGRU, and 2 times higher than LSTM and GRU. This implies that the significant performance gain of GCQN

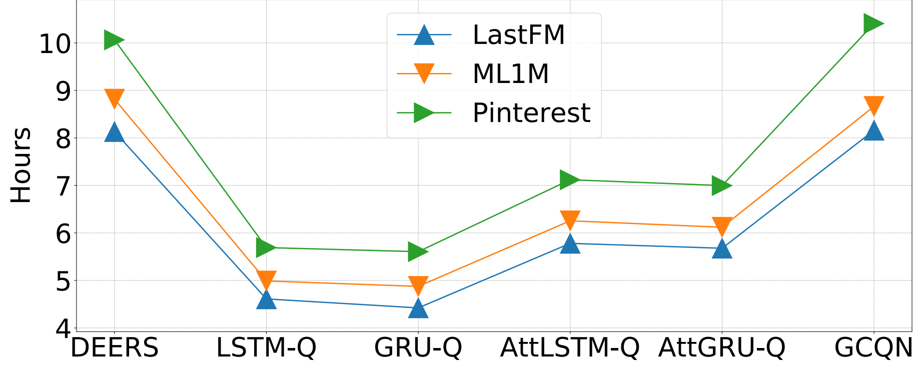


Figure 4.4: Comparison of running time of reinforcement learning based methods. The training cost of GCQN is close to DEERS, nearly 1.5 times higher than AttLSTM and AttGRU, and 2 times higher than LSTM and GRU. This indicates that the significant performance gain of GCQN is achieved with acceptable cost.

Table 4.3: The results of GCQN with different aggregators.

AGG	**LastFM**		**ML1M**		**Pinterest**	
	$T = 10$	$T = 20$	$T = 10$	$T = 20$	$T = 10$	$T = 20$
mean	0.459	0.398	0.687	0.644	0.227	0.211
pool	0.458	0.400	0.691	0.648	0.221	0.211
att	**0.464**	**0.404**	**0.697**	**0.658**	**0.232**	**0.215**

is achieved with reasonable cost[10].

Impact of Aggregator. To study the impact of aggregator on the performance of GCQN, we design two variants of GCQN by replacing its attention-based aggregator AGG^{att} with the mean-based one AGG^{mean} and the pooling-based one AGG^{pool} proposed in [38]. The comparison results in Table 4.3 show that the proposed AGG^{att} outperforms the others by a significant margin. This verifies the efficacy of using attention mechanisms to capture different neighbors' features, instead of treating them equally.

Impact of Neighborhood Sample Size L. We investigate the influence of

[10]In the propose models, we use -greedy algorithm to select actions and perform off-policy learning. Millions of transitions are needed for updating the models in order to achieve good performance. We will explore more to further improve the sample efficiency in future work.

Table 4.4: The results of GCQN with different sample size L.

L	LastFM		ML1M		Pinterest	
	$T = 10$	$T = 20$	$T = 10$	$T = 20$	$T = 10$	$T = 20$
2	0.458	0.401	0.694	0.652	0.226	0.209
5	0.461	0.403	0.695	0.652	0.228	**0.217**
10	**0.464**	**0.404**	**0.697**	**0.658**	**0.232**	0.215
20	0.458	0.401	0.693	0.651	0.230	0.212

neighborhood sample size L on the performance of GCQN. The results of different $L \in \{2, 5, 10, 20\}$ are shown in Table 4.4. We observe that GCQN with $L = 10$ achieves the best performance (except for the case of $T = 20$ on Pinterest). This is because a too small L is not able to incorporate enough neighborhood information, while a too large L might bring noisy information.

4.5 Conclusions and Discussions

Our method is partly inspired by the recent advancements of graph convolutional networks (GCNs). GCNs are a class of neural network architectures for processing graph-structured data, which aim to learn meaningful representations of nodes given a graph [161, 151]. To achieve this goal, spectral GCNs [11, 27, 56] perform "graph convolutions" based on the spectral graph theory, while non-spectral GCNs [30, 92, 38] directly take operations (e.g., weighted average) on the local neighborhood of a node. As shown in these prior works, the learned GCN-based representations are able to significantly improve the model's performance in a variety of tasks such as node classification and link prediction. Recently, researchers have applied GCNs to a variety of recommendation domains and demonstrated appealing performance. PinSage [153] extends the GraphSAGE algorithm [38] to the pin-board bipartite graph for Pinterest recommender system. Some works apply GCNs to the user-item bipartite graph in collaborative filtering recommender systems [7, 91, 142].

Moreover, a number of GCN-based methods have been proposed for session-based recommendations [150, 115], social recommender systems [32, 149], and knowledge-graph-aware recommender systems [139, 137].

However, these existing GCN-based methods are essentially supervised learning based approaches, which can only learn static, passive, and shortsighted predictive models for the single-step recommendation problems. *In this work, we make the first attempt to effectively leverage GCNs to design RL approaches to learn dynamic, proactive, and farsighted recommendation policies.*

We propose a novel idea that designs graph-structured states and actions for RL recommendation agents. In particular, the proposed graph-structured state seamlessly integrates the structural information in user-item graph and the sequential information in user-consumed item sequence, such that the factors of personalization and collaboration are effectively embedded. Both types of information is crucial to the learning of personalized recommendation policies. To implement the idea, we develop an effective end-to-end RL agent, termed Graph Convolutional Q-network (GCQN). GCQN is able to approximate the optimal action-value function based on the inputs of graph-structured representations, by successfully leveraging GCN and GRU to exploit the structural and sequential information, respectively. We have empirically validated the proposed idea, as well as the developed GCQN, by conducting solid experiments on real-world datasets.

Our approach is a generic framework for graph-structure reinforcement learning, which can be readily extended with many existing techniques. For instance, other GCN designs such as fastGCN [16] can be incorporated into GCQN to further improve its performance. Also, the action-value predicting module in GCQN can be replaced by the dueling network architectures [144], in order to improve policy learning in those cases where many similar-valued actions exist. Moreover, the state-representation learning module in GCQN can be transferred into a policy net-

work under the policy gradient framework [71], in order to handle continuous action spaces.

Currently, we have not made a systematical comparison between UDQN (the MF-based model) and GCQN (the GCN-based model). However, we did make a rough comparison between them when we recently conducted Research Work 4. We found that: GCQN performs better on sparse datasets such as Amazon, but worse on dense datasets such as MovieLens, compared to UDQN.

Chapter 5

Social Attentive Deep Q-network: Improving Personalization and Collaboration via Social Attention

In this work, we propose an effective way to address the issues of data sparsity and cold-start of existing RL-based recommendation agents, by leveraging the available social network among users to promote policy learning. Specifically, we develop a Social Attentive Deep Q-network (SADQN) agent to learn recommendation policies based on the preferences of both individual users and their social neighbors, by successfully utilizing social attention to model the social influence between them. SADQN further models the factors of personalization and collaboration, on the basis of UDQN, by utilizing a combined action-value prediction module that consists of a personal action-value function and a social action-value function. Moreover, we propose an enhanced variant of SADQN, termed SADQN++, to model the complicated and diverse trade-offs between personal preferences and social influence for all involved users, making the agent more powerful and flexible in learning optimal policies. The solid experimental results on real-world datasets demonstrate that the proposed SADQNs remarkably outperform the state-of-the-art agents, with reasonable computation cost.

5.1 Introduction

The majority of research on recommendation algorithms is based on supervised learning, which focuses on learning accurate predictive models from historical feedback data for only single-step recommendations [1, 100, 58, 156]. Most of these recommendation approaches cannot provide a satisfactory solution to the multi-step interactive recommendation problem in real-world scenarios. To address this issue, a potential way is to use RL, which aims at learning an agent that can auto-control its behavior in an environment, in order to achieve a goal [119]. By integrating both RL and deep neural networks, deep RL agents have shown human-level or even better performance in solving many complex decision making problems such as playing Atari [90, 124] and Go [110]. Recently, a number of researchers incorporated the ideas and techniques of deep RL into recommender systems, and proposed several novel recommendation algorithms which have shown great potential in a variety of recommendation domains [160, 158, 157, 19]. Compared to traditional recommenders, a notable merit of RL agents is that they are able to actively discover users' interests through user-agent interactions and recommend items that may bring maximal future rewards.

Successful as they are, the existing RL based approaches only exploit user-item feedback data, whose recommendation performance may be greatly reduced when data sparsity and cold-start occur. Fortunately, with the emergence of online social networks such as Twitter, additional social information of users is usually available to the recommender. According to the social influence theory, users are influenced by others in the social network, leading to the homophily effect that social neighbors may have similar preferences [9, 4]. Thus, it is a potential way to improve the quality of recommendations by leveraging available social networks, which has been widely studied and demonstrated in traditional social recommendation domains [82, 80, 53,

107, 122, 152]. However, most of the existing social recommendation models cannot be directly extended to RL based systems, as they are based on the paradigm of supervised learning.

To the best of our knowledge, this paper makes the first attempt to improve the performance of deep RL based recommenders, by effectively utilizing available social networks. In the context of deep RL, the performance of many agents is determined by the estimation of the *long-term rewards*, e.g., action-values, which indicate, in the long run, how many benefits the agent will obtain if it recommends the items at current time [119]. Similar to the prediction of *short-term rewards* (e.g., ratings) in traditional recommendation domains, it is biased and inefficient to estimate the action-values based on only users' own preferences, due to the issues of cold-start and data sparsity. Thus, we propose to leverage social neighbors' preferences to promote the estimation of action-values.

To implement this idea, we develop a novel deep RL agent, termed Social Attentive Deep Q-network (SADQN). The key idea is that we estimate the action-values by a linear combination of two action-value functions, the *personal action-value function* Q^P and the *social action-value function* Q^S (see Figure 5.2). Intuitively, Q^P estimates action-values based on target user's personal preferences, as most of the existing methods do. In contrast, Q^S is able to estimate action-values based on his/her social neighbors' preferences, by utilizing an *attention mechanism* to model the influence from different social neighbors to target user. By integrating both functions, SADQN is able to learn recommendation policies that take advantage of both personal preferences and social influence.

While SADQN is straightforward and easily understandable, the simple linear combination of function approximators Q^P and Q^S is not able to model the complicated and diverse trade-offs between personal interests and social influence for all involved users, which limits the performance in approximating the optimal action-

value function Q^*. To handle this challenge, we propose an enhanced variant of SADQN, termed SADQN++ (see Figure 5.3), to fuse Q^P and Q^S more deeply by learning appropriate trade-offs from data autonomously. More specifically, we leverage Q^P and Q^S to learn some relevant hidden representations from personal and neighbors' preferences. Then, we employ additional neural layers to summarize valuable features from these hidden representations, and predict the final action-value based on the summarized features. This way provides more flexibility to SADQN++ in modeling the trade-offs, leading to stronger capability in approximating the optimal action-value function. As a result, SADQN++ is able to learn the optimal policies more effectively.

We empirically validate the performance of the proposed SADQNs by conducting solid experiments on three real-world datasets. The results show that they remarkably outperform four state-of-the-art deep RL agents that fail to consider social influence, as well as several traditional recommendation methods. In particular, the relative improvements of SADQN++ against the best performing baseline are at least larger than 8.5% for cold-start recommendation, and 3.5% for warm-start recommendation. More importantly, the significant improvements of SADQNs over the state-of-the-art agents are accomplished with reasonable computation cost.

5.2 Preliminaries

5.2.1 Problem Formulation

We consider a recommender system with user set $\mathcal{U} = \{1, ..., m\}$ and item set $\mathcal{I} = \{1, ..., n\}$. Let $R \in \mathbb{R}^{m \times n}$ denote the user-item feedback matrix, where $R_{ia} = 1$ if user i gives a positive feedback on item a (clicks, watches, etc.), and $R_{ia} = 0$ otherwise.

We focus on the task of recommending items in sequential user-recommender interactions, which can be formulated as a standard RL problem [119]. Specifically,

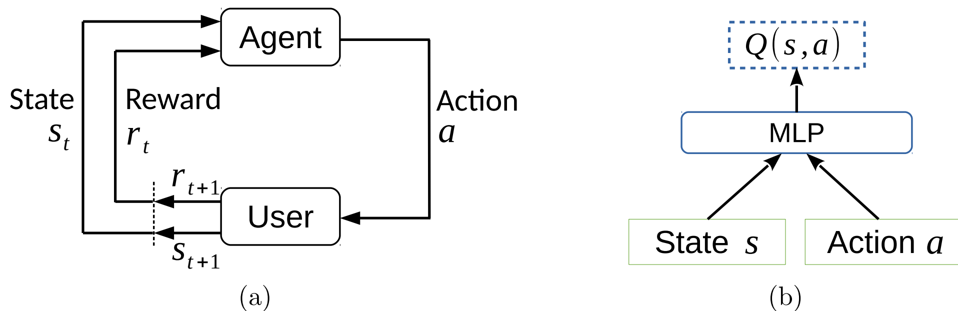


Figure 5.1: (a) The user-agent interaction in RL based recommender systems. (b) The basic architecture of DQN.

an agent (recommender) and an environment (target user i) interact at discrete time steps (see Figure 5.1a). At each time step t, the agent observes the environment's state s_t (representing the current preferences of user i), and accordingly takes an action (item) a_t based on its policy (probability distributions over actions given states). One time step later, as a consequence of its action, the agent receives a reward r_{t+1} (R_{ia_t}) and next state s_{t+1} from the environment. The goal of the agent is to maximize the cumulative reward it receives in T interactions.

5.2.2 Reinforcement Learning

Normally, the agent's policy is a mapping from states to actions, $a = \pi(s)$, or probability distributions over actions given states, $\pi(a|s)$. To learn the optimal policy, the basic idea of many RL methods is to estimate the action-value (Q) function, which is formally defined as:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum\nolimits_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right], \tag{5.1}$$

where γ is the discount factor that balances the importance between future rewards and immediate rewards. It indicates, in the long run, how good it is to take action a in state s while following policy π in future steps. The optimal Q function, $Q^*(s, a) = \max_\pi Q^\pi(s, a)$, obeys an important identity known as the Bellman opti-

mality equation [6]:

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') \Big| s,a\right]. \tag{5.2}$$

Based on the above equation, a popular RL algorithm, Q-learning [145], estimates a Q function (i.e., a lookup table) via the following online update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[y - Q(s,a)\right], \tag{5.3}$$

where $y = r + \gamma \max_{a'} Q(s',a')$ is usually called Q-learning target, α is the learning rate, and (s,a,r,s') denotes a transition the agent experiences during its interactions with the environment. The estimated Q function will converge to Q^* when the number of iterations goes to infinity [119]. The greedy policy w.r.t. Q^* will be an optimal policy π^*.

5.2.3 Deep Q-network

The classical Q-learning algorithm must maintain a lookup table of all state-action pairs, which cannot handle complex RL tasks with enormous state and action spaces. Moreover, it estimates the action-value function separately for each sequence, which has no generalization ability to deal with unseen states and actions [119].

To address these issues, a common practice is to approximate the optimal action-value function by using a function approximator, i.e., $Q(s,a;\theta) \approx Q^*(s,a)$, where θ denote the weights to be learned [119]. For instance, Deep Q-network (DQN) [90] employs a deep neural network as the function approximator. The basic form of DQN is illustrated in Figure 5.1b. It is actually a multilayer perceptron (MLP), which takes state s and action a as input, and outputs the predicted action-value $Q(s,a)$.

The Q-network $Q(s,a;\theta)$ can be trained by performing Q-learning updates based on the agent's experiences (s,a,r,s'). Specifically, the loss function that needs to be

minimized is defined as:

$$\mathcal{L}(\theta) = \mathbb{E}_{s,a,r,s'}\left[(y - Q(s,a;\theta))^2\right], \tag{5.4}$$

where $y = r + \gamma \max_{a'} Q(s', a'; \theta^-)$ is the target for current iteration, and θ^- are the network weights from previous iteration, which are held fixed when optimizing the loss function. In practice, rather than optimizing the full expectations, a more convenient way is to perform stochastic gradient descent (SGD) on sampled transitions (s, a, r, s') [89, 90]:

$$\theta \leftarrow \theta + \alpha\left[y - Q(s,a;\theta)\right]\nabla_\theta Q(s,a;\theta). \tag{5.5}$$

5.3 Social Attentive Deep Q-networks

As mentioned before, it is biased and inefficient to estimate the optimal Q^* function (corresponding to an optimal recommendation policy) only based on individual users' own feedbacks, due to the issues of cold-start and data sparsity. In this section, we propose a novel class of deep RL agents, which are able to estimate Q^* more effectively, by leveraging the available social network among users. Let $S \in \mathbb{R}^{m\times m}$ be the adjacency matrix of the social network, where $S_{ij} = 1$ if user i has a positive relation to user j (follows, trusts, etc.), and $S_{ij} = 0$ otherwise. Let $\mathcal{N}(i) = \{j : S_{ij} = 1\}$ denote the set of social neighbors whom user i trusts/follows.

We assume that the state s_t is a f-dimensional feature vector $U_i^t \in \mathbb{R}^f$, denoting the real-time preferences of target user i at time step t. For each user $j \in \mathcal{U}$, there is a f-dimensional feature vector $U_j \in \mathbb{R}^f$, denoting the overall preferences of user j observed in advance. For each item (action) $a \in \mathcal{I}$, there is also a f-dimensional feature vector $V_a \in \mathbb{R}^f$, denoting the overall features of item a. The feature matrices $U \in \mathbb{R}^{f\times m}$ and $V \in \mathbb{R}^{f\times n}$ are trained by standard matrix factorization [59] together with negative sampling [97] based on the historical feedback data R, which are held fixed during the user-agent interactive recommendation process. The target user's

vector U_i^t (i.e., state s_t) is initialized as the trained U_i at time step $t = 0$, and is updated by performing online matrix factorization on the real-time feedback data R_{ia_t} for each time step t:

$$U_i^{t+1} \leftarrow U_i^t + \alpha \left[\left(R_{ia_t} - (U_i^t)^{\mathrm{T}} V_{a_t} \right) V_{a_t} - \lambda U_i^t \right], \tag{5.6}$$

where α is the learning rate, and λ is the L_2 regularization parameter.

In what follows, we first present a basic Social Attentive Deep Q-network (SADQN), describing how to estimate action-values based on the preferences of both individual users and their social neighbors. Then, we propose an enhanced variant of SADQN, SADQN++, which integrates personal preferences with social influence more deeply. Lastly, we describe the training algorithm of SADQNs.

5.3.1 SADQN: A Linear Fusion Model

The basic idea behind SADQN is that the action-value $Q(s_t, a)$ is estimated by a linear combination of two action-value functions $Q^P(s_t, a)$ and $Q^S(s_t, a)$, which denote the *personal action-value function* and the *social action-value function*, respectively. Intuitively, Q^P estimates the action-values based on target user i's personal real-time preferences, i.e., U_i^t. In contrast, Q^S estimates the action-values based on his/her social neighbors' pre-observed preferences, i.e., U_j for $j \in \mathcal{N}(i)$, which are correlated with U_i^t according to the social influence theory.

The architecture of SADQN is illustrated in Figure 5.2. The right part of SADQN is the personal action-value function approximator Q^P, which is a standard 4-layer MLP. It takes the concatenation of user vector U_i^t (i.e., the features of state s_t) and item vector V_a (i.e., the features of action a) as input, followed by two fully connected (FC) layers, and outputs the personal action-value $Q^P(s_t, a)$. In our experiments, each FC layer consists of 256 neurons with ReLU activation. Therefore, the specific architecture of MLP^P is $2f \to 256 \to 256 \to 1$.

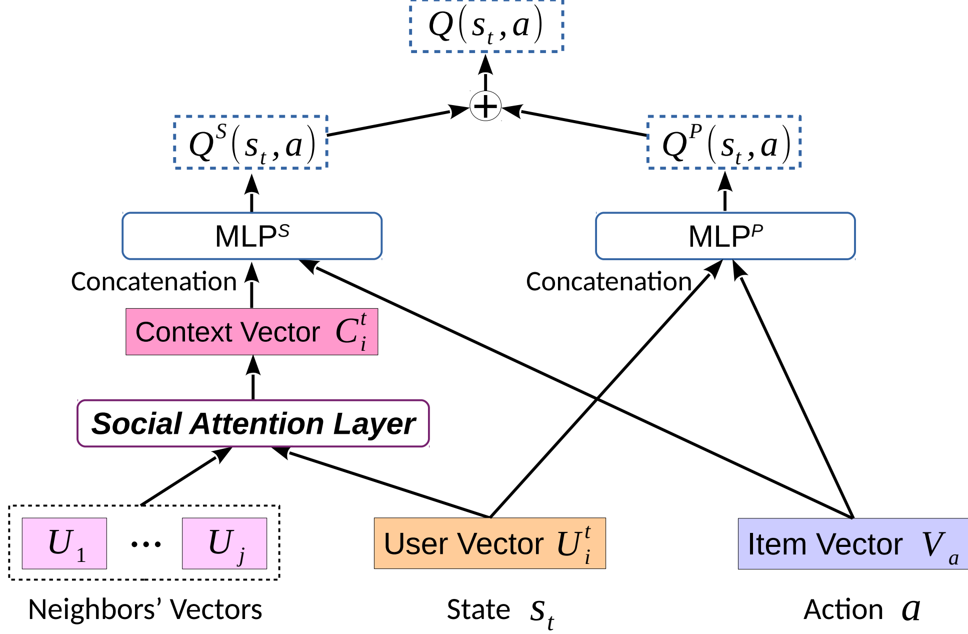


Figure 5.2: SADQN: a linear fusion model.

The left part of SADQN is the social action-value function approximator Q^S, in which the core is a *social attention* (SA) layer. The goal of the SA layer is to select influential social neighbors for target user i at time step t, and summarize the neighbors' features to a context vector C_i^t. Then, the concatenation of context vector C_i^t and item vector V_a is used to feed MLP^S with the same architecture, which will output the social action-value $Q^S(s_t, a)$.

Specifically, we compute the context vector C_i^t by the following procedure. We employ the CONCAT attention mechanism to calculate the *attention coefficient* of target user i and his/her social neighbor $j \in \mathcal{N}(i)$:

$$e_{ij}^t = \text{ReLU}(\mathbf{w}^{\text{T}} \cdot \text{CONCAT}(U_i^t, U_j)), \tag{5.7}$$

where $\mathbf{w} \in \mathbb{R}^{2f}$ is the weight vector of a single-layer feedforward network. The attention coefficient e_{ij}^t indicates the social influence strength of user j to user i at time step t. Similar to [128], we also compute the attention coefficient of user i and himself/herself by: $e_{ii}^t = \text{ReLU}(\mathbf{w}^{\text{T}} \cdot \text{CONCAT}(U_i^t, U_i^t))$. Then, we compute the

normalized attention coefficients α_{ij}^t by softmax function:

$$\alpha_{ij}^t = \frac{\exp(e_{ij}^t)}{\sum_{k \in \mathcal{N}(i)_+} \exp(e_{ik}^t)}, \tag{5.8}$$

where $\mathcal{N}(i)_+ = \mathcal{N}(i) \cup \{i\}$. Finally, we obtain the context vector C_i^t by:

$$C_i^t = \alpha_{ii}^t U_i^t + \sum\nolimits_{j \in \mathcal{N}(i)} \alpha_{ij}^t U_j. \tag{5.9}$$

In our experiments, we also tried several different attention mechanisms to compute the context vector C_i^t. For example, the attention coefficient e_{ij}^t can be computed by the simple DOT product:

$$e_{ij}^t = \mathrm{DOT}(U_i^t, U_j). \tag{5.10}$$

However, the performance of this approach showed no significant difference with the one in Equation 5.7. Moreover, we also implemented a single-layer graph attention network (GAT) [128] to compute C_i^t. Unfortunately, it did not show comparable performance against the above two approaches (see Table 5.3 for comparison).

To summarize, the action-value $Q(s_t, a)$ estimated by SADQN is formally defined as:

$$\begin{aligned}
\mathbf{h}_1^P &= \mathrm{CONCAT}(U_i^t, V_a),\\
\mathbf{h}_2^P &= \mathrm{ReLU}(\mathbf{W}_2^P \cdot \mathbf{h}_1^P + \mathbf{b}_2^P),\\
\mathbf{h}_3^P &= \mathrm{ReLU}(\mathbf{W}_3^P \cdot \mathbf{h}_2^P + \mathbf{b}_3^P),\\
Q^P(s_t, a) &= (\mathbf{w}_4^P)^{\mathrm{T}} \cdot \mathbf{h}_3^P + \mathrm{b}_4^P; \qquad (5.11)\\
\mathbf{h}_1^S &= \mathrm{CONCAT}(C_i^t, V_a),\\
\mathbf{h}_2^S &= \mathrm{ReLU}(\mathbf{W}_2^S \cdot \mathbf{h}_1^S + \mathbf{b}_2^S),\\
\mathbf{h}_3^S &= \mathrm{ReLU}(\mathbf{W}_3^S \cdot \mathbf{h}_2^S + \mathbf{b}_3^S),\\
Q^S(s_t, a) &= (\mathbf{w}_4^S)^{\mathrm{T}} \cdot \mathbf{h}_3^S + \mathrm{b}_4^S; \qquad (5.12)\\
Q(s_t, a) &= (1-\beta) Q^P(s_t, a) + \beta Q^S(s_t, a). \qquad (5.13)
\end{aligned}$$

Here, $\mathbf{h}_l^P$, $\mathbf{W}_l^P$ and $\mathbf{b}_l^P$ denote the outputs, trainable weights and biases of l-th layer of MLP^P, respectively. Similar notations are used for MLP^S. $\beta \in [0, 1]$ controls the trade-off between personal preferences and social influence.

The network parameters of SADQN can be trained by performing the Q-learning updates in Equation 5.5. Note that if we only use Q^P or Q^S to estimate the action-values, SADQN will reduce to a *pure personal model* SADQN^P (which is equivalent to the basic DQN model shown in Figure 5.1b) or a *pure social model* SADQN^S.

5.3.2 SADQN++: A Deep Fusion Model

In the previous SADQN model, the personal approximator Q^P and the social approximator Q^S are simply fused by a linear combination at the output level. While this approach is straightforward and easily understandable, such a shallow fusion might limit the performance of Q-network in approximating the optimal action-value function Q^*. For example, the trade-offs between personal interests and social influence may vary considerably, for different users i at different time steps t, or even on different items a.

As such, using the same trade-off parameter β (in Equation 5.13) for all situations is obviously inappropriate. In fact, we tested different $\beta \in \{0.2, 0.5, 0.8\}$ in our experiments. But the overall performance shows no significant difference, which implies that a good trade-off for one situation may be improper for another. On the other hand, it is infeasible to search or learn an optimal value of β for every situation, e.g., for each state-action pair (s_t, a).

To address this issue, we propose an enhanced variant of SADQN, termed SADQN++, to more deeply fuse the two approximators Q^P and Q^S by autonomously learning good trade-offs from data via additional neural layers.

The architecture of SADQN++ is illustrated in Figure 5.3, which is quite similar to SADQN, with only differences in the last few layers. Rather than using Q^P and

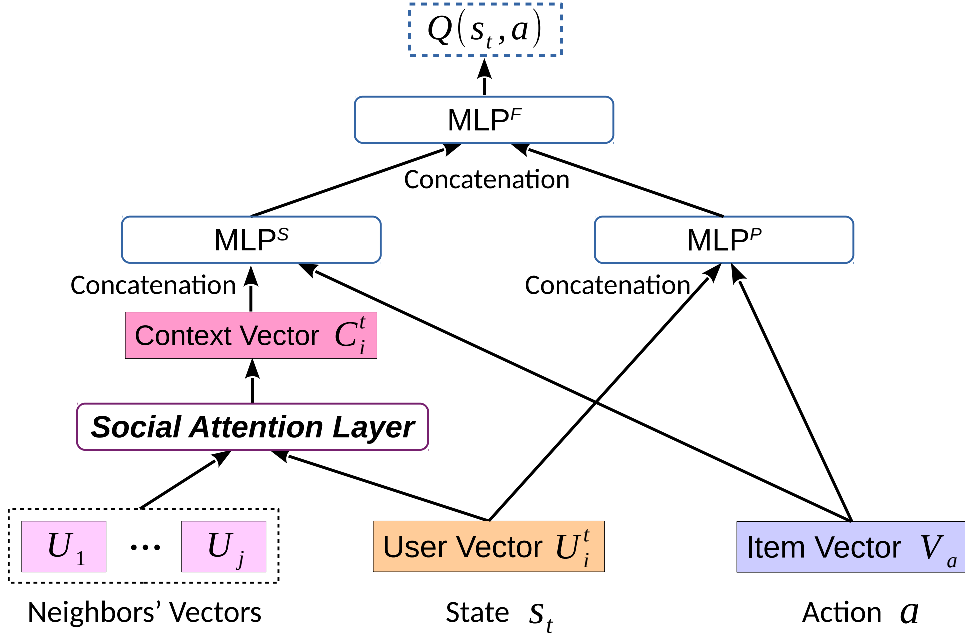


Figure 5.3: SADQN++: a deep fusion model.

Q^S to directly estimate action-values from personal and neighbors' preferences, in SADQN++, we leverage them to learn some hidden vector representations that are relevant to the estimation of action-values. More specifically, MLP^S (MLP^P) in SADQN++ is a 3-layer MLP with the architecture of $2f \rightarrow 256 \rightarrow 256$, which will output the *social hidden representation* (*personal hidden representation*). Then, we employ MLP^F with the architecture of $512 \rightarrow 256 \rightarrow 1$, to autonomously summarize valuable features from both hidden representations, and to predict the final action-value $Q(s_t, a)$ based on the summarized features.

This way of deep fusion provides more flexibility to the agent in approximating optimal action-values, making it possible to capture the complicated and diverse trade-offs between personal preferences and social influence for all involved users in real-world scenarios. Thus, SADQN++ is able to learn optimal policies in a more effective way. More formally, the action-value $Q(s_t, a)$ estimated by SADQN++ is

given by:

$$\begin{aligned}
\mathbf{h}_1^P &= \text{CONCAT}(U_i^t, V_a), \\
\mathbf{h}_2^P &= \text{ReLU}(\mathbf{W}_2^P \cdot \mathbf{h}_1^P + \mathbf{b}_2^P), \\
\mathbf{h}_3^P &= \text{ReLU}(\mathbf{W}_3^P \cdot \mathbf{h}_2^P + \mathbf{b}_3^P); \qquad (5.14) \\
\mathbf{h}_1^S &= \text{CONCAT}(C_i^t, V_a), \\
\mathbf{h}_2^S &= \text{ReLU}(\mathbf{W}_2^S \cdot \mathbf{h}_1^S + \mathbf{b}_2^S), \\
\mathbf{h}_3^S &= \text{ReLU}(\mathbf{W}_3^S \cdot \mathbf{h}_2^S + \mathbf{b}_3^S); \qquad (5.15) \\
\mathbf{h}_1^F &= \text{CONCAT}(\mathbf{h}_3^P, \mathbf{h}_3^S), \\
\mathbf{h}_2^F &= \text{ReLU}(\mathbf{W}_2^F \cdot \mathbf{h}_1^F + \mathbf{b}_2^F), \\
Q(s_t, a) &= (\mathbf{w}_3^F)^{\text{T}} \cdot \mathbf{h}_2^F + \text{b}_3^F. \qquad (5.16)
\end{aligned}$$

Here, $\mathbf{h}_l^P$, $\mathbf{W}_l^P$ and $\mathbf{b}_l^P$ denote the outputs, trainable weights and biases of l-th layer of MLP^P, respectively. Similar notations are used for MLP^S and MLP^F.

5.3.3 Training Algorithm

To train SADQNs (i.e., the proposed Q-networks), we employ the popular Q-leaning algorithm [145]. We do not adopt the training techniques of *experience replay* and *target network* used by the original DQN [90], as they are not able to improve the Q-learning performance for our task. To make the Q-network Q converge well, sufficient transitions (s, a, r, s') of all possible states and actions are needed for Q-learning updates [119]. To this end, we propose a particular training scheme that enables the agent to collect transitions based on the feedback data of all training users.

Specifically, in each episode, we uniformly sample a user i from training set $\mathcal{U}_{train}$ as the current target user, which will interact with the agent and generate corresponding states and rewards. To ensure exploration, in each state s_t, the agent uses a ϵ-greedy policy that selects a greedy action $a_t = \arg\max_a Q(s_t, a)$ with probabil-

Algorithm 5.1: Training SADQNs

Input: $\mathcal{U}_{train}$, R, the trained feature matrices U, V
Output: the trained Q-network Q

1 Initialize Q with random weights
2 **for** $episode = 1, N$ **do**
3   Uniformly sample a target user i from $\mathcal{U}_{train}$
4   Set initial state $s_0 = U_i^0 = U_i$
5   **for** $t = 0, T-1$ **do**
6   Choose the ϵ-greedy item a_t w.r.t. $Q(s_t, a)$
7   Present a_t to user i and receive feedback R_{ia_t}
8   Get U_i^{t+1} according to **Equation 5.6**
9   Set reward $r_{t+1} = R_{ia_t}$ and state $s_{t+1} = U_i^{t+1}$
10   Update Q's weights on $(s_t, a_t, r_{t+1}, s_{t+1})$ according to **Equation 5.5**

ity $1 - \epsilon$ and a random action with probability ϵ. The full algorithm for training SADQNs is presented in Algorithm 5.1. The training process could last for any number of episodes as long as the Q-network has not converged. At the testing stage, the trained agent can be used to make real-time interactive recommendations for any new user j. It only needs to interact with user j, observe state s_t (U_j^t), and recommend the greedy item $a_t = \arg\max_a Q(s_t, a)$ at each time step t.

Computational Complexity Analysis

In the inner for-loop in Algorithm 5.1, the computation time is mainly taken in computing Q-values for available items (line 6), updating user vector U_i^{t+1} (line 8), and updating Q-network (line 10). The cost of computing Q-values is $O(n|\theta|)$, where $|\theta|$ is the number of Q-network weights and n is the number of all items. The cost of updating Q-network is $O(|\theta|)$. The cost of updating U_i^{t+1} is $O(f)$, where f is the dimensionality of latent feature space. Therefore, the time complexity of the training algorithm of SADQNs is $O(NT(n|\theta| + f))$, where N is the number of episodes and T is the number of time steps. Similarly, we can derive that the cost of performing T-step interactive recommendations for a new user is $O(T(n|\theta| + f))$.

Table 5.1: The Statistics of Datasets

Statistics	**LastFM**	**Ciao**	**Epinions**
#users	1,874	7,260	23,137
#items	2,828	11,166	23,585
#observed user feedbacks	71,411	147,799	461,982
density of feedbacks	1.35%	0.18%	0.08%
#observed social relations	25,174	110,715	372,205
density of relations	2.03%	0.28%	0.09%

5.4 Experiments

To validate the performance of the proposed SADQNs, we conduct extensive experiments on real-world datasets. In this section, we first introduce our experimental setup, followed by presenting the experimental results and analysis.

5.4.1 Experimental Setup

Datasets

We employ three publicly available datasets: LastFM[1] [12], Ciao[2] [121], and Epinions[3] [86] for our experiments. All the datasets contain a user-item feedback matrix and a user-user social network. As we consider the recommendation problem with implicit feedback, we convert the values of all observed feedbacks to 1. Besides, we remove the users or items that have fewer than 5 feedbacks, so as to ensure that there is enough data for training and testing. The basic statistics of the obtained datasets are shown in Table 5.1.

[1] https://grouplens.org/datasets/hetrec-2011/

[2] https://www.cse.msu.edu/~tangjili/trust.html

[3] http://www.trustlet.org/downloaded_epinions.html

Evaluation Methodology

To conduct experiments on interactive recommendations, we assume that the observed feedbacks in the datasets are unbiased, as proposed in [69, 159]. Similar to [26], we randomly choose 1000 unobserved (i, a) pairs of user i as the negative feedbacks. During the T-step interactive recommendation process, the agent is forced to pick items from the available set that consists of the 1000 negative items and the observed positive items.

We adopt two popular evaluation metrics *Hit Ratio* (HR) and *Normalized Discounted Cumulative Gain* (NDCG). The HR metric indicates the ratio of positive items among the T recommended items, which is defined as:

$$\text{HR} = \frac{1}{T}\sum_{t=0}^{T-1} R_{ia_t}, \tag{5.17}$$

where a_t is the item recommended at time step t, and $R_{ia_t} = 1$ ($R_{ia_t} = 0$) if a_t is a positive (negative) item w.r.t. user i.

The NDCG metric is computed by following procedure. At each time step t, a ranking list of the available items is produced according to the agent's predictions (e.g., Q-values). The DCG(t) value is calculated by:

$$\text{DCG}(t) = \sum_{j=1}^{k} \frac{2^{R(j)} - 1}{\log_2(1 + j)}, \tag{5.18}$$

where k is the length of the ranking list, j denotes the rank position in the ranking list, and $R(j)$ is the ground-truth feedback of the j-th item. The NDCG(t) is calculated by:

$$\text{NDCG}(t) = \frac{\text{DCG}(t)}{Z}, \tag{5.19}$$

where Z is the DCG(t) value of the optimal ranking list sorted by ground-truth feedbacks. The final NDCG is obtained by averaging the NDCG(t) values for time

steps $t = 0, ..., T-1$. We truncate the ranking list at 10 to compute the NDCG@10 values, and set $T = 20$ for evaluation.

We conduct experiments for two different recommendation scenarios: cold-start and warm-start. In the *cold-start setting*, we assume that the agent has no feedback data of target user at time step $t = 0$, i.e., at the beginning of the interactive recommendation process. We randomly choose 10% users who have at least 20 positive feedbacks as the testing set $\mathcal{U}_{test}$, and others as the training set $\mathcal{U}_{train} = \mathcal{U} \backslash \mathcal{U}_{test}$, which are used to test and train the agent, respectively. The data of $\mathcal{U}_{train}$ is also used to train the feature matrices U and V. To fit the cold-start scenario, in each episode of training phase, the target user's vector U_i^0 (i.e., initial state s_0) is set to a randomized vector rather than the trained U_i (see line 4 in Algorithm 5.1).

In the *warm-start setting*, we assume that the agent already has 10 positive feedbacks of target user at time step $t = 0$. We select the users who have at least 30 positive feedbacks as the target set $\mathcal{U}_{tar}$, and others as the pre-training set $\mathcal{U}_{pre} = \mathcal{U} \backslash \mathcal{U}_{tar}$. We randomly choose 10% users from $\mathcal{U}_{tar}$ as the testing set $\mathcal{U}_{test}$, and the remaining as the training set $\mathcal{U}_{train} = \mathcal{U}_{tar} \backslash \mathcal{U}_{test}$. We use R_{pre}, R_{train} and R_{test} to denote the data of $\mathcal{U}_{pre}$, $\mathcal{U}_{train}$ and $\mathcal{U}_{test}$, respectively. Then, for each user in $\mathcal{U}_{train}$ ($\mathcal{U}_{test}$), we randomly choose 10 positive feedbacks and move them from R_{train} (R_{test}) to R_{pre}. The final data R_{pre}, R_{train} and R_{test} is used to train the feature matrices, train and test the agent, respectively. In this warm-start scenario, the target user's vector U_i^0 (state s_0) already captures some preference information of him/her.

For both cold-start and warm-start settings, the testing HR or NDCG@10 value for an ideal agent will be 1. Besides, for each setting and each dataset, we conduct each experiment on 5 data splits obtained with different random seeds, and calculate the mean and standard deviation of the 5 groups of results for evaluation.

Baselines

We comparatively evaluate the proposed **SADQNs** against a variety of baselines, which are listed below.

1. **DQN** [90], a state-of-the-art deep RL agent, which is originally designed for playing Atari.

2. **DDQN** [124], a state-of-the-art deep RL agent, which extends DQN with double Q-learning [40].

3. **DRN** [160], a state-of-the-art deep RL agent for news recommendation, which is based on Dueling DQN [144] that estimates the action-values via both value function and advantage function.

4. **DEERS** [158], a state-of-the-art deep RL agent for recommendation, which utilizes Gated Recurrent Units (GRU) to learn state features from both positive and negative item click sequences.

5. **LinUCB** [69], a representative contextual bandit algorithm for news recommendation.

6. **SoRec** [82], a representative social matrix factorization method, which factorizes both feedback matrix and social network simultaneously.

7. **TrustMF** [152], a representative social matrix factorization method, which models the mutual influence between trusters and trustees in the trust network.

8. **MF** [59], a conventional matrix factorization model, which only exploits the feedback matrix.

9. **Impact** [87], an active learning method, which picks the item that has highest impact on other items, where the impact is computed on the user-item bipartite

graph.

10. **Pop**, a popularity-based method, which picks the item which has most positive feedbacks given by users.

To make the baselines applicable to our task, we adopt the same state/action features and training scheme of SADQNs for DQN, DDQN, DRN and LinUCB, and use a negative sampling technique of uniform distribution for MF, SoRec and TrustMF. We also adopt the same hidden layers of the personal action-value function of SADQN, i.e., two FC layers of 256 units with ReLU activation, for DQN, DDQN and both the value and advantage functions of DRN, which lead to better performance. For DEERS, we adopt the same architecture suggested in the original reference [158]. Moreover, to make a fair comparison, we set the feature dimensionality $f = 64$ for all methods (excluding Pop, Impact and DEERS). Other parameters are tuned based on cross-validation, which are set as follows: the regularization parameter $\lambda = 0.01$, the learning rate for updating feature vectors $\alpha = 0.01$, the learning rate for updating Q-networks $\alpha = 0.0001$, the discount factor $\gamma = 0.5$, and the ϵ-greedy parameter $\epsilon = 0.1$.

It is important to notice that, our interactive recommendation task is distinctly different from session-based recommendation [44] or temporal social recommendation [117]. In their works, the recommender is developed to passively learn a predictive model from time series data such as $a_1, a_2, ..., a_{t-1}$, and to predict the next item a_t that may appear in the series. In contrast, our RL agent is designed to actively learn a recommendation policy from user-agent interactions, and to provide items that may optimize the cumulative reward in a T-step recommendation process. As such, those Recurrent Neural Network based models [44, 117] are inapplicable to our task, and are not compared in our experiments.

Moreover, although there are many social recommendation models proposed very

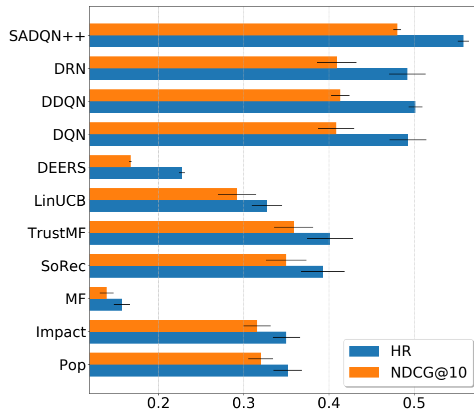
recently, they cannot be applied to interactive recommendation unless making critical extensions to them. Unfortunately, most of them cannot be easily extended, such as the ones proposed in [74, 14, 149, 32, 17]. Thus, we only compare with two representative social recommendation models, SoRec and TrustMF, which are very flexible and can be extended to fit our task by online learning and negative sampling.

5.4.2 Performance Comparison against Baselines

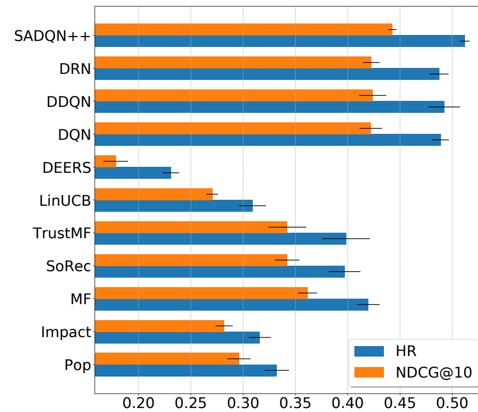
We now compare the performance of SADQN++ (the best performing variant of SADQN) against the baseline methods. Figure 5.4 shows the comparison results in terms of the mean (bar) and standard deviation (line) of HR and NDCG@10 metrics for cold-start and warm-start recommendations, respectively. From these results, we have the following main findings.

For cold-start recommendation, the proposed SADQN++ model shows the best performance in terms of both metrics on all datasets. It remarkably outperforms the four deep RL agents DQN, DDQN, DRN and DEERS that fail to consider social influence, as well as other types of baselines. For example, its improvements in HR metric against the best performing baseline are 11.19%, 9.47% and 8.88% on LastFM, Ciao and Epinions datasets, respectively. These results not only verify the capability of SADQN++, but also demonstrate that social influence plays a fundamental role in improving the performance of deep RL recommenders.

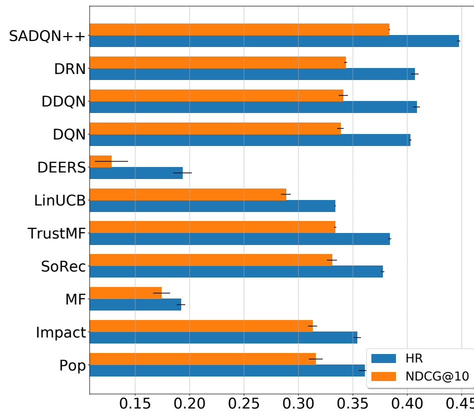
The DQN, DDQN and DRN agents show the second-class performance, while DEERS performs almost the worst which implies that it might be inappropriate to our task. The traditional matrix factorization model MF shows poor performance, as no feedback data is available at time step $t = 0$, while the social recommendation models SoRec and TrustMF demonstrate much better performance. Besides, the popularity method Pop and active learning method Impact are also competitive baselines in the cold-start setting.



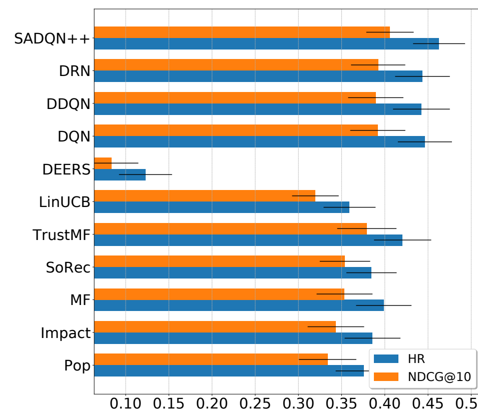
(a) Cold-start, LastFM



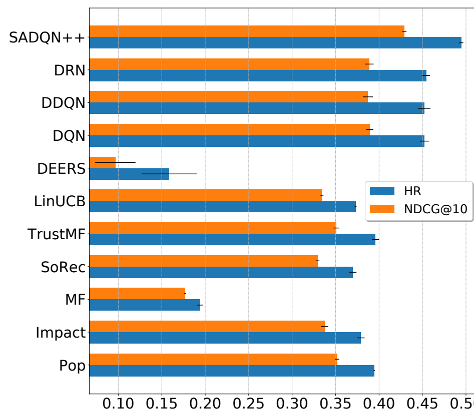
(b) Warm-start, LastFM



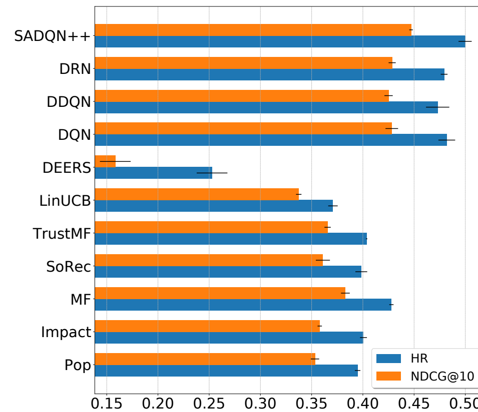
(c) Cold-start, Ciao



(d) Warm-start, Ciao



(e) Cold-start, Epinions



(f) Warm-start, Epinions

Figure 5.4: Performance comparison against baselines on both **cold-start** and **warm-start** recommendations. The mean (bar) and standard deviation (line) of HR and NDCG@10 metrics on all three datasets are shown. The proposed SADQN++ model shows the best performance in all cases.

For warm-start recommendation, the proposed SADQN++ model also shows significantly better performance than the competitors. Specifically, its improvements in HR (NDCG@10) metric against the best performing baseline are 3.9%, 3.6% and 3.7% (4.4%, 3.5% and 4.3%) on LastFM, Ciao and Epinions datasets, respectively. The results of baselines show similar trends compared to cold-start setting, with one exception of the MF model, which shows competitive performance in the warm-start setting.

5.4.3 The Impact of Social Influence

To quantitatively analyze the impact of social influence, here, we make a more detailed comparison among the four variants of SADQN. They are the pure personal model SADQN^P (which is equivalent to the DQN baseline), the pure social model SADQN^S, the linear fusion model SADQN and the deep fusion model SADQN++, respectively. The comparison results of the mean and standard deviation of HR and NDCG@10 metrics are reported in Table 5.2. The relative improvements of SADQN^S, SADQN and SADQN++ against SADQN^P are shown in the brackets, which tell us clearly how social influence increasingly improves the performance when we model it from shallowly to deeply. Also, the best result in each case is highlighted. From this part of results, we observe the following main points.

The deep fusion model SADQN++ performs much better than others. In particular, its relative improvements against the personal model SADQN^P are at least larger than 9.0% for cold-start recommendation, and 3.5% for warm-start recommendation. Also, the improvements in cold-start setting show an interesting trend that they are totally consistent with the densities of social relations in the datasets. More specifically, the order of the improvements in HR (NDCG@10) metric on LastFM, Ciao and Epinions datasets is 13.2% > 11.0% > 9.3% (17.5% > 13.2% > 10.2%), same as the order of relation densities 2.03% > 0.28% > 0.09% (see Table 5.1). This

Table 5.2: The Experimental Results of Different Variants of SADQN

	Model	Cold-start Recommendation		Warm-start Recommendation	
		HR	**NDCG@10**	**HR**	**NDCG@10**
LastFM	$SADQN^P$	0.492±0.021	0.408±0.021	0.488±0.008	0.422±0.010
	$SADQN^S$	0.523±0.012 (+6.2%)	0.464±0.013 (+13.7%)	0.476±0.009 (-2.4%)	0.413±0.006 (-2.1%)
	SADQN	0.543±0.008 (+10.3%)	0.472±0.009 (+15.7%)	0.497±0.013 (+1.6%)	0.429±0.006 (+1.8%)
	SADQN++	**0.557±0.006 (+13.2%)**	**0.480±0.004 (+17.5%)**	**0.511±0.004 (+4.7%)**	**0.442±0.003 (+4.9%)**
Ciao	$SADQN^P$	0.402±0.001	0.338±0.003	0.446±0.031	0.391±0.031
	$SADQN^S$	0.417±0.005 (+3.6%)	0.362±0.004 (+7.1%)	0.447±0.034 (+0.2%)	0.391±0.030 (+0.0%)
	SADQN	0.424±0.007 (+5.4%)	0.364±0.001 (+7.6%)	0.451±0.030 (+1.1%)	0.397±0.027 (+1.3%)
	SADQN++	**0.447±0.001 (+11.0%)**	**0.383±0.001 (+13.2%)**	**0.462±0.030 (+3.6%)**	**0.406±0.027 (+3.6%)**
Epinions	$SADQN^P$	0.452±0.005	0.389±0.004	0.481±0.008	0.428±0.006
	$SADQN^S$	0.446±0.001 (-1.2%)	0.397±0.001 (+2.0%)	0.468±0.004 (-2.8%)	0.416±0.004 (-2.8%)
	SADQN	0.478±0.001 (+5.7%)	0.415±0.001 (+6.7%)	0.487±0.001 (+1.2%)	0.435±0.001 (+1.6%)
	SADQN++	**0.494±0.002 (+9.3%)**	**0.429±0.002 (+10.2%)**	**0.499±0.006 (+3.7%)**	**0.447±0.001 (+4.4%)**

demonstrates that, more social relations SADQN++ exploits, more benefits it will obtain.

The linear fusion model SADQN performs second-best in all cases, and shows similar trends with SADQN++.

The pure social model SADQN^S also performs better than SADQN^P in cold-start setting (except for the case of HR metric on Epinions dataset), but shows worse performance in warm-start setting (in most cases). This implies that, when the social network data is extremely sparse, or when the user-item feedback data is sufficient, the SADQN^S model purely using social influence cannot achieve desirable recommendation performance.

5.4.4 Model Analysis

Comparison of Different Attention Mechanisms

We now compare the performance of different attention mechanisms discussed in Section 5.3.1. We conduct an experiment to compare three SADQN agents which adopt the attention mechanisms GAT [128], DOT (Equation 5.10), and CONCAT (Equation 5.7, i.e., the default one used by SADQN), respectively. This experiment is only conducted in cold-start recommendation setting. The comparison results in terms of both HR and NDCG@10 metrics on all three datasets are shown in Table 5.3, where the bold font indicates the best result in each case. The two attention mechanisms CONCAT and DOT perform very closely, and both outperform GAT.

Run-time Analysis

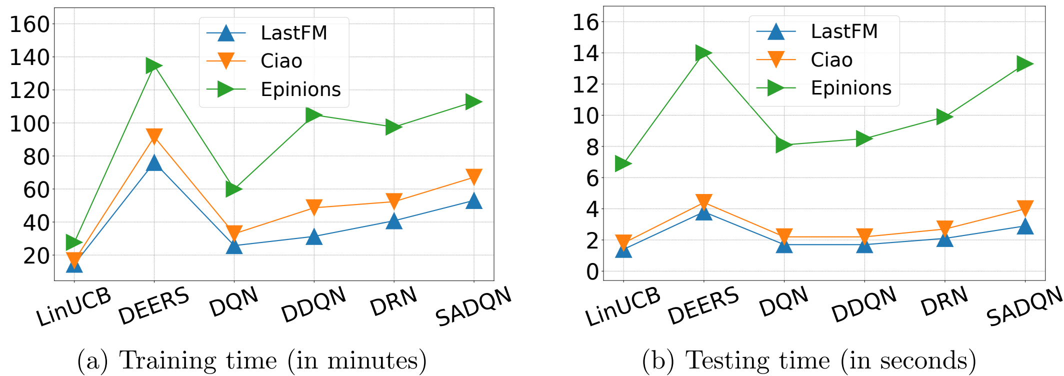
As analyzed previously, the time complexity of the training algorithm of SADQNs is $O(NT(n|\theta| + f))$, where N is the number of episodes for training, T is the number of time steps in each episode, n is number of items in the dataset, $|\theta|$ is the number of Q-network weights, and f is the dimensionality of latent feature space. Since f

Table 5.3: The Results of SADQN with Different Attention Mechanisms

Dataset	Attention	HR	NDCG@10
LastFM	GAT	0.5191±0.0114	0.4393±0.0127
	DOT	**0.5438±0.0036**	0.4717±0.0084
	CONCAT	0.5435±0.0076	**0.4724±0.0091**
Ciao	GAT	0.4179±0.0060	0.3567±0.0082
	DOT	**0.4256±0.0031**	**0.3670±0.0014**
	CONCAT	0.4247±0.0072	0.3644±0.0011
Epinions	GAT	0.4667±0.0018	0.4033±0.0005
	DOT	0.4755±0.0016	0.4142±0.0030
	CONCAT	**0.4782±0.0012**	**0.4154±0.0013**

is usually a small constant in practice (f=64 in our experiments), we can rewrite the time complexity as $O(Ln|\theta|)$, where $L = NT$ denotes the number of Q-learning updates that is needed to make the Q-network converge. In other words, the time cost for training SADQN agent is related to three basic variables: the size of Q-network, the size of data, and the convergence of Q-learning algorithm. Similarly, we can derive that the time cost for testing SADQN agent is related to the size of Q-network and the size of data.

Here, we show the run-time of SADQN for cold-start recommendation on a single-GPU machine. The time cost for warm-start recommendation would be similar. We also show the time costs of RL-based baselines for comparison. The non-RL methods are not compared here, as their time costs are much lower and can be ignored in comparison to RL-based methods. Both the training time (in minutes) and the testing time (in seconds) of one run of the compared methods on all three datasets are shown in Figure 5.5. Overall, the testing costs of all compared methods are quite low, which demonstrates that they are able to perform real-time online recommendations. For the training cost, the proposed SADQN is lower than DEERS, close to DDQN and DRN, but higher than DQN and LinUCB. The results demonstrate that our



(a) Training time (in minutes) (b) Testing time (in seconds)

Figure 5.5: Comparison of the run-time of RL-based methods.

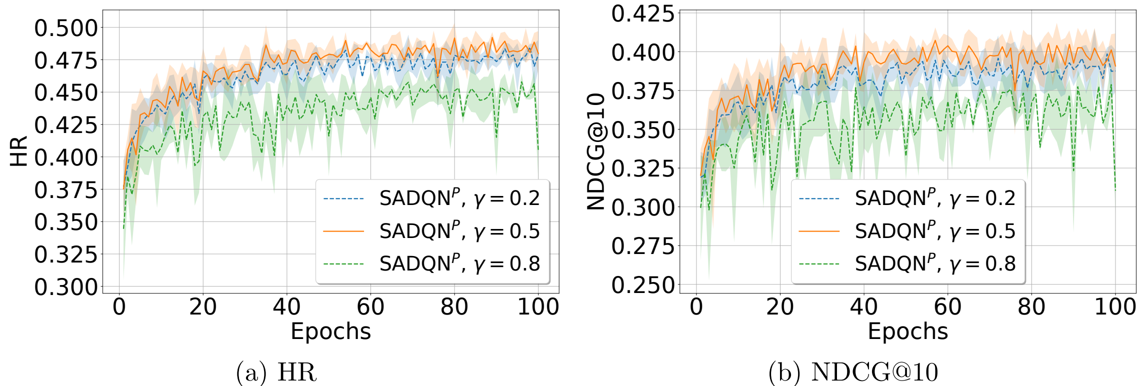
approach is able to improve the performance of the state-of-the-art RL agents, with acceptable additional cost.

Parameter Analysis

In this subsection, we conduct experiments to show how the proposed SADQN agents behave with different settings of some important hyperparameters. When analyzing a specific parameter, others are fixed to the default settings. More specifically, we use the simplest variant of SADQN, SADQN^P, as an example to conduct the analysis. In addition, these experiments are only conducted in cold-start setting using LastFM dataset.

Effect of the Discount Factor γ. The discount factor $\gamma \in [0, 1]$ balances the trade-off between future rewards and immediate rewards when estimating Q-values. The selection of its value usually depends on the natures of particular RL tasks [119].

Here, we vary γ in $\{0.2, 0.5, 0.8\}$ to compare the performance of SADQN^P. The learning curves in terms of both mean (line) and standard deviation (shadow) of HR and NDCG@10 metrics are shown in Figure 5.6, where each epoch corresponds to 50k Q-learning updates in Algorithm 5.1. Note that the learning curves are plotted from epoch 1. The results demonstrate two main points. First, SADQN^P with $\gamma = 0.5$ achieves the best performance. Second, SADQN^P with a lower γ learns more stably



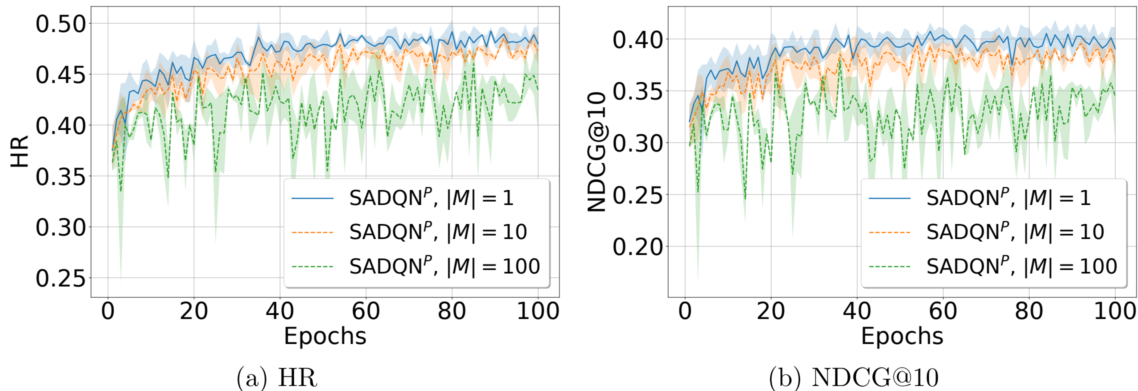
(a) HR (b) NDCG@10

Figure 5.6: Effect of the discount factor γ on the performance of SADQN^P.

than a higher one.

Effect of the Experience Replay. The training trick of experience replay was first proposed by [72]. To perform experience replay, a memory M is needed to store the transitions that the agent collects during the user-agent interactive process. At each time step, the Q-network is updated based on a minibatch of transitions sampled from M, rather than based on the currently observed transition (see line 10 in Algorithm 5.1). The experience replay has shown appealing performance improvements in some RL tasks such as playing Atari [90].

Here, we validate whether and how much the experience replay will benefit SADQN^P for a totally different recommendation task. To do this, we vary the memory size $|M| \in \{1, 10, 100\}$ to check how the agent's performance changes. Note that $|M| = 1$ (i.e., the default setting used by SADQNs) implies that the agent actually does not adopt experience replay. The comparison results are shown in Figure 5.7, which clearly tell us the same trend: SADQN^P with $|M| = 1$ achieves the best performance, and its performance will be consistently reduced when the memory becomes larger. The implies, different from traditional tasks, the use of experience replay will significantly reduce the agent's performance for our recommendation task.



(a) HR (b) NDCG@10

Figure 5.7: Effect of the experience replay on the performance of SADQN^P.

5.5 Conclusions and Discussions

Deep RL has been successfully applied to recommender systems, but still heavily suffer from data sparsity and cold-start, leading to insufficient modeling of personalization and collaboration. In this work, we address these issues by strengthening the state-of-the-art deep RL recommenders with social influence among users. We develop a class of Social Attentive Deep Q-networks (SADQNs) to estimate action-values based on the preferences of both individual users and social neighbors, by successfully utilizing an attention mechanism to model the social influence between them. In particular, we proposed an enhanced variant of SADQN, termed SADQN++, which is able to model the complicated trade-offs between personal preferences and social influence for all users, making the agent more powerful and flexible in learning optimal policies.

We conducted extensive experiments on three real-world datasets to verify the effectiveness and efficiency of the proposed SADQNs. The results have demonstrated that social influence plays a fundamental role in improving the recommendation performance of deep RL, especially in the cold-start recommendation scenarios. More importantly, the significant improvements of SADQNs over the state-of-the-art agents are accomplished with acceptable computation cost.

Chapter 6

Personalized Deep Q-network: Integrating Personalized Network Architecture with Collaborative Learning Objective

In this work, we provide a more explicit and effective way to model personalization and collaboration in RL-based recommendation agents. We develop an end-to-end agent, term Personalized Deep Q-network (PDQN), which is able to learn a fully personalized policy that makes recommendations based on both the user-specific information of a particular user and the general information of state shared by all users. More specifically, PDQN estimates personalized action-values based on the linear combination of two action-value functions: the user-specific action-value function and the general action-value function that emphasize on modeling personalization and collaboration, respectively. Furthermore, we propose a novel collaborative Q-learning objective to further model the collaborations between similar neighbors, which is able to make the PDQN agent more effective in approximating the optimal personalized policy. We show that PDQN achieves significant performance gains over the existing methods on several real-world datasets.

6.1 Introduction

Reinforcement learning (RL) [119] is a promising approach to recommender systems, which aims to build a recommendation agent that is able to adaptively recommend potentially interesting items to users in a sequential manner. Compared to non-RL recommendation engines, a key advantage of RL agents is that they are able to not only capture users' dynamic preferences via continuous user-agent interactions, but also learn farsighted policies that achieve maximal long-term rewards from users. Recently, researchers introduced the techniques of deep RL to design novel recommendation agents with good generalization ability and scalability. These modern RL agents, which use neural networks to approximate the optimal action-value functions or policies, have shown great potential in a variety of recommendation domains ranging from news feeds to E-commerce sites [160, 158, 19, 20].

Despite their successes, however, there is a fundamental problem that has been rarely noticed and investigated in these prior works. That is, *how to effectively model the factors of personalization and collaboration in an RL recommendation agent, in order to provide high-quality personalized recommendations for the entire user community?* Both personalization and collaboration are crucial factors to the learning of personalized recommendation policy that can optimize the overall profit of the recommender system. Unfortunately, we find that the prior works have paid little attention to the modeling of personalization and collaboration. Most of the existing RL recommendation agents [106, 120, 160, 158, 19, 20, 163] follow the same learning paradigm that treats all involved users as a single virtual user and directly learns a unified policy $\pi(s)$ that outputs actions dependent only on the state s (usually, a sequence of user-consumed items). Although this learning paradigm is straightforward and can be easily implemented, it only models very limited personalization and very coarse-grained collaboration. To be more specific, the existing learning paradigm is

not able to model the diverse long-term preferences of different users if they are in the same current state s (usually appears at earlier time steps), since the unified policy $\pi(s)$ will make the same action in state s. On the other hand, it treats all the other users equally as the "collaborators" of each target user, which is not able to effectively learn collaborative preferences from his/her truly similar neighbors. Some works [76, 75] make an extension to the learning paradigm by incorporating the additional user id into the state s. However, it is still very challenging to simultaneously model both of the personalization and collaboration, i.e., the differences and similarities between users, into a single representation of state.

In this work, we provide a more effective way to model personalization and collaboration in RL recommendation agents. We propose a novel learning paradigm, named personalized reinforcement learning (PRL), to learn a unified but personalized policy $\pi(u, s)$ that outputs actions based on both the user-specific information of user u and the general information of state s shared by all users. More specifically, to reduce the difficulty in modeling personalization and collaboration, the personalized policy $\pi(u, s)$ is divided into two sub-policies $\pi_1(u)$ and $\pi_2(s)$, such that $\pi(u, s) = f(\pi_1(u), \pi_2(s))$, where f denotes a function that integrates the two sub-policies. Intuitively, $\pi_1(u)$ emphasizes on modeling personalization and indicates how to pick actions given the user-specific information of user u, while $\pi_2(s)$ focuses on modeling collaboration and indicates how to pick actions given the general information of state s. In this way, PRL is able to simultaneously model the personalized long-term preferences of each user and the collaborative relationships between different users, more effectively.

To implement the idea of PRL, we develop an end-to-end value-based agent, term Personalized Deep Q-network (PDQN), which is able to learn a personalized action-value function $Q(u, s, a)$. The greedy policy following $Q(u, s, a)$ with be a desirable personalized policy $\pi(u, s)$. To be more specific, PDQN estimates

$Q(u, s, a)$ via a linear combination of two action-value functions: the user-specific action-value function $Q_U(u)$ and the general action-value function $Q_G(s, a)$, i.e., $Q(u, s, a) = Q_U(u) + Q_G(s, a)$. Intuitively, $Q_U(u)$ is able to model personalization by predicting a user-specific action-value based on the user-specific information of user u, while $Q_G(s, a)$ is able to model collaboration by predicting a general action-value based on the general information of state s shared by all users. As a result, the learned personalized action-value function $Q(u, s, a)$ is able to more effectively produce high-quality personalized and collaborative recommendations for the entire user community. Furthermore, we propose a novel collaborative Q-learning objective that contains a particular collaborative regularizer to further model collaboration, which is able to make PDQN to approximate the optimal personalized policy more effectively. We refer to the PDQN agent trained with the collaborative Q-learning objective as PDQN-cr.

We apply the proposed PDQN and PDQN-cr agents to the interactive recommendation task on a number of real-world datasets. The results demonstrate that PDQN and PDQN-cr are able to achieve significant performance margins over the state-of-the-art recommendation agents.

6.2 Preliminaries

6.2.1 Interactive Recommendation

Suppose we have a recommender system that involves a set of m users $\mathcal{U} = \{1, ..., m\}$ and a set of n items $\mathcal{I} = \{1, ..., n\}$. Let $Y \in \mathbb{R}^{m \times n}$ be the observed user-item feedback matrix , where $y_{ui} = 1$ if user u gives a positive feedback on item i (clicking, watching, etc.), and $y_{ui} = 0$ otherwise. We consider an interactive recommendation problem defined as follows. A recommendation agent and a target user u interact at discrete time steps $t = 0, ..., T - 1$. At each time step t, the agent recommends an item i_t to

user u according to its current observations, then receives a feedback y_{ui_t} given by user u. After considering the feedback information, the agent updates its observations and recommends a new item i_{t+1} at time step $t+1$. The goal of the agent is to recommend T items that can receive most positive feedbacks from user u, i.e., $\max \sum_{t=0}^{T-1} y_{ui_t}$.

6.2.2 Markov Decision Process

We study the interactive recommendation problem under the framework of reinforcement learning (RL) [119]. The agent-user interaction in recommendation can be naturally modeled as the agent-environment interaction in RL (see Figure 1.1). This gives rise to an episodic RL task, where in each episode, the agent interacts with the environment (corresponding to the target user u) at discrete time steps $t = 0, ..., T-1$. At each time step t, the agent observes a state s_t of the environment, then takes an action a_t according to its policy π (indicating how to choose actions given states). One time step later, as a result of its action, the agent receives a numerical reward r_{t+1} and a new state s_{t+1} from the environment.

More formally, the environment can be mathematically described by a Markov decision process (MDP), that is, a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ defined as follows:

- $\mathcal{S}$ **is the state space.** The state $s_t \in \mathcal{S}$ denotes any information that is useful for making decisions (i.e., recommending items to user u) at time step t. In this work, we define the state s_t as a sequence of k latest observed item-feedback pairs before time step t, i.e., $s_t = \{(a_{t-k}, y_{ua_{t-k}}), ..., (a_{t-1}, y_{ua_{t-1}})\}$, where a_j denotes the recommended item at time step j, and y_{ua_j} denotes the corresponding feedback given by user u. In particular, $s_0 = \{(a_{-k}, y_{ua_{-k}}), ..., (a_{-1}, y_{ua_{-1}})\}$ is the initial state that consists of a sequence of k pre-observed item-feedback pairs of user u before the interactive recommendation process, and $s_T = \phi$ is the terminal state in which the interactive recommendation process ends. Clearly, we have $s_{t+1} \backslash \{(a_t, y_{ua_t})\} = s_t \backslash \{(a_{t-k}, y_{ua_{t-k}})\}, \forall t = 0, ..., T-2$.

- $\mathcal{A}$ **is the action space.** We define $\mathcal{A}$ as the set of all items, i.e., $\mathcal{A} = \mathcal{I}$, where each action $a \in \mathcal{A}$ is corresponding to a unique item. In each state s_t, an action a_t can be taken from the set of available actions $\mathcal{A}(s_t)$, which is defined recursively: $\mathcal{A}(s_t) = \mathcal{A}(s_{t-1}) \backslash \{a_{t-1}\}$ for $t \neq 0$, and $\mathcal{A}(s_0) = \mathcal{A}$. This definition forces the agent to always choose a new item that has not been recommended before each time step t.

- $\mathcal{P} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ **is the state-transition function.** $\mathcal{P}^a_{ss'} = Pr[s_{t+1} = s'|s_t = s, a_t = a]$ denotes the probability that the environment transits to state s' after receiving action a in state s. In the context of recommendation, the explicit specifications of the transition probabilities are unknown. The agent can only observe specific state transitions by interacting with the environment step by step during each episode.

- $\mathcal{R} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow \{-1, 1\}$ **is the reward function.** $\mathcal{R}^a_{ss'} = E[r_{t+1}|s_t = s, a_t = a, s_{t+1} = s']$ denotes the expected immediate reward the environment generates after the transition from state s to s' due to action a. In the context of recommendation, the immediate reward of executing an action a only depends on the feedback given by user u. Therefore, we define $\mathcal{R}^a_{ss'} = 1$ (-1) if $y_{ua} = 1$ (0). This definition will give a strong penalty to the agent if it recommends a negative item.

It is worth noting that this MDP satisfies the **Markov property**:

$$
\begin{aligned}
& Pr[s_{t+1} = s', r_{t+1} = r | s_0, a_0, r_1, ..., s_{t-1}, a_{t-1}, r_t, s_t, a_t] \\
& = Pr[s_{t+1} = s', r_{t+1} = r | s_t, a_t],
\end{aligned}
\tag{6.1}
$$

for all s', r, and histories $s_0, a_0, r_1, ..., s_{t-1}, a_{t-1}, r_t, s_t, a_t$. This provides well guarantees for applying standard RL methods to solve the MDP [119], that is, to estimate

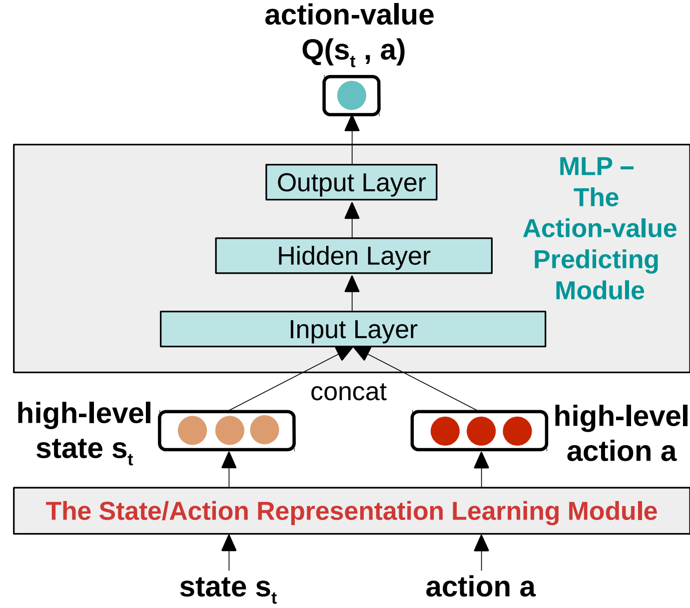


Figure 6.1: The basic DQN agent.

an optimal policy π^* that can maximize the cumulative reward the agent received during the interactive recommendation process.

6.2.3 Deep Q-network

To handle the MDP that involves large state and action spaces, a well-adopted approach is the Deep Q-network (DQN) [90], which approximates the optimal action-value function Q^* (corresponding to an optimal policy π^*) by using a neural network function approximator Q (known as Q-network), i.e., $Q(s, a; \theta) \approx Q^*(s, a)$ [72, 119]. Here, θ denotes the weight vector of the Q-network that can be learned with Q-learning based algorithms [145]. To be more specific, we derive a basic DQN model for the interactive recommendation problem, as shown in Figure 6.1. This DQN consists of a standard multilayer perceptron (MLP), which takes high-level vector representations of state s and action a as input, and outputs the predicted action-value $Q(s, a; \theta)$.

The DQN can be trained by performing Q-learning updates based on the agent's

experiences collected from the interactions with the environment, i.e., the transition data in the form of tuple (s, a, r, s'). Specifically, the loss function that needs to be minimized is:

$$\mathcal{L}(\theta) = \mathbb{E}_{s,a,r,s'}\left[(y - Q(s, a; \theta))^2\right], \tag{6.2}$$

where $y = r + \gamma \max_{a'} Q(s', a'; \theta^-)$ is the Q-learning target for the current iteration, and θ^- is the network's weight vector from the previous iteration, which is held fixed when optimizing the loss function. In practice, rather than optimizing the full expectations in the above loss function, a more convenient way is to perform stochastic gradient descent (SGD) on sampled transitions (s, a, r, s') [89, 90]:

$$\theta \leftarrow \theta + \alpha \left[y - Q(s, a; \theta)\right] \nabla_\theta Q(s, a; \theta). \tag{6.3}$$

6.3 Personalized Deep Q-network

6.3.1 Motivation and Idea

So far, we have described a standard RL task and its MDP formulation for the target user u's interactive recommendation process, and have presented a basic DQN agent that is able to learn recommendation policies by interacting with the MDP. Unfortunately, this is far from the desirable solution that can really work to make meaningful interactive recommendations in practice. A real-world recommender system usually involves multiple users, where each user u's interactive recommendation process is actually a unique MDP, denoted by MDP_u. This gives rise to a non-typical multi-MDP RL task, as illustrated in Figure 1.2, where the agent needs to interact with the multiple MDPs of all users. Since the involved users usually have diverse preferences and behaviors over time steps, their MDPs may vary remarkably in terms of both state-transition functions and reward functions. In other words, similar users' MDPs may have relatively close dynamics, but dissimilar users' MDPs may have distinctly

different dynamics. As a result, it is a very difficult challenge for the agent to learn an effective policy that makes suitable recommendations for all involved users.

To tackle this challenge, we argue that a desirable RL recommendation agent should capture simultaneously the two key factors: personalization and collaboration. Personalization is probably the most important factor of any real-world recommender system that attracts user attention and improves user experience. It implies that the agent should model the specific personalized characteristics of each user as much as possible, in order to learn a personalized policy that produces interesting user-specific recommendations to diverse users. *Collaboration* means that the agent should model the underlying collaborative relationships (e.g., behavioral similarities) between different but similar users, in order to learn a collaborative policy that makes more effective recommendations for the entire user community. Collaboration is also crucial to building an effective RL-based recommender system to work in practice because of two reasons. First of all, due to the severe issue of data sparsity in recommendation[1], the agent cannot interact with a single user's MDP continuously to collect enough and representative transition data to update the policy towards an optimal one. Moreover, collaboration offers important clues to the agent to make novel recommendations for each target user, by discovering his/her unknown interests from similar neighbors (i.e., the collaborators), especially in the cold-start stage. In summary, personalization and collaboration offer the agent large opportunities to reveal the differences and similarities among the multiple MDPs of users, respectively. As a result, by properly modeling both the personalization and collaboration of users, a desired agent is able to learn more effective recommendation policies for the Multi-MDP task.

Unfortunately, most of the existing RL recommendation agents follow the same learning paradigm that treats all involved users as a single virtual user and directly

[1]That is to say, most users only have feedback on a very small fraction of items.

learns a unified policy $\pi(s) : \mathcal{S} \to \mathcal{A}$ based on the transition data collected uniformly from the multiple MDPs[2]. Although the existing RL learning paradigm is simple and can be easily implemented, it fails to effectively model the personalization and collaboration factors. To be more specific, it only models very limited and implicit personalization in terms of state representations, which is not able to capture the diverse long-term preferences of different users if they are in the same current state s (usually appears at earlier time steps in the interactive process), since the unified policy $\pi(s)$ will make the same action in state s. On the other hand, it models very coarse-grained collaboration that treats all the other users equally as the collaborators of each target user, which is not able to effectively discover similar preferences from his/her truly collaborative neighbors.

Another potential approach to solving the multi-MDP task is to leverage multi-agent reinforcement learning (MARL) to construct the same number of agents. Each agent aims to learn an independent policy $\pi_u(s)$ from each target user u's MDP and make corresponding recommendations. Theoretically, MARL is able to learn a set of fully personalized recommendation policies. However, it is practically infeasible in real-world recommender systems that usually involve a very large number of users, since the training of the multiple MARL agents is extremely time-consuming. Moreover, how to effectively model collaboration is also an open problem in MARL.

In this work, we propose a novel learning paradigm, named personalized reinforcement learning (PRL), which is able to model both the personalization and collaboration factors in a more effective way. Different from the existing RL and MARL learning paradigms, as illustrated in Figure 6.2, PRL aims to learn a unified but personalized policy $\pi(u, s) : \mathcal{U} \times \mathcal{S} \to \mathcal{A}$ that outputs actions based on both the user-specific information of user u and the general information of state s shared by

[2] In this work, we use the deterministic policy $\pi(s)$ (i.e., a mapping from states) as an example to present our core ideas and methods, which are also applicable to the cases of stochastic policy $\pi(a|s)$ (i.e., probabilities over actions given states).

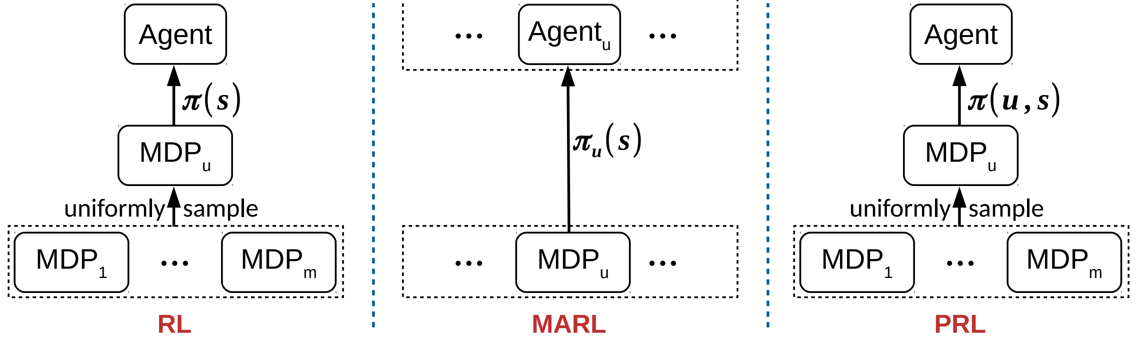


Figure 6.2: The three learning paradigms for the multi-MDP task: the existing RL and MARL, and our proposed PRL.

all users. Apparently, the learned policy $\pi(u, s)$ is able to produce fully personalized recommendations for diverse users, even in the same state s. On the other hand, since the user-specific term properly embeds the differences of users' preferences in the policy $\pi(u, s)$, PRL is able to model more fine-grained collaboration between different users (even those dissimilar users) by effectively exploiting the shared general information of state s.

6.3.2 The PDQN Model

To implement the idea of PRL, we take DQN as the basis to design a novel end-to-end agent, termed Personalized Deep Q-network (PDQN). As shown in Figure 6.3, PDQN predicts the personalized action-value $Q(u, s_t, a)$ of each triplet (u, s_t, a) by taking the input of user u, state s_t, and action a. By performing continuing Q-learning updates based on the transition data $(u, s_t, a_t, r_{t+1}, s_{t+1})$ collected from the MDP_u of all users u, PDQN is able to approximate the optimal personalized action-value function $Q^*(u, s_t, a)$. The greedy policy following $Q^*(u, s_t, a)$, i.e., taking action $a_t = \arg\max_a Q^*(u, s_t, a)$ given user u and state s_t, will be an optimal personalized policy $\pi^*(u, s_t)$.

To be more specific, PDQN is the linear combination of two action-value functions: the *user-specific* action-value function Q_U and the *general* action-value func-

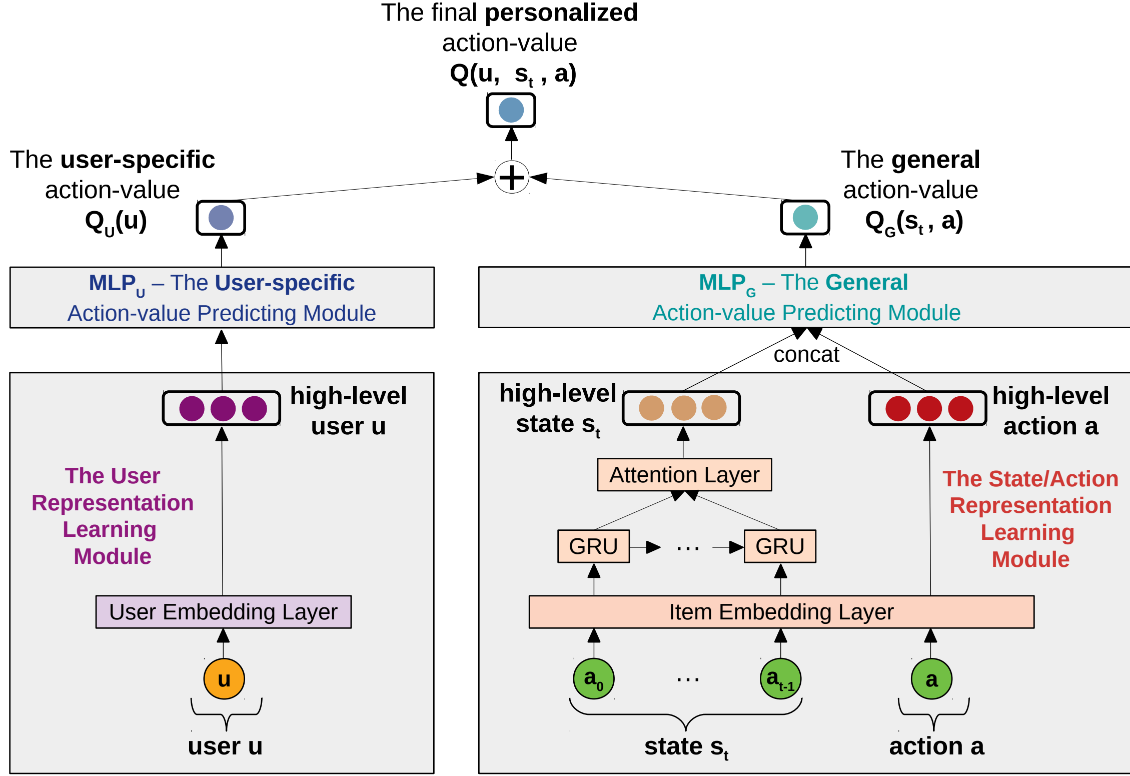


Figure 6.3: The proposed PDQN agent.

tion Q_G. That is, the final personalized action-value $Q(u, s_t, a)$ is given by:

$$Q(u, s_t, a) = Q_U(u) + Q_G(s_t, a). \tag{6.4}$$

Intuitively, Q_U models the user-specific characteristics of each user u's MDP, while Q_G models the general characteristics shared by all users' MDPs. We will elaborate on them in the following.

The User-specific Action-value Function Q_U

The goal of Q_U is to predict the user-specific action-value $Q_U(u)$ for each user u, independent to particular states and actions. Q_U consists of two basic modules: the user representation learning module that aims to learn a high-level vector representation of user u, and the user-specific action-value predicting module that attempts to estimate the user-specific action-value $Q_U(u)$ based on the learned high-level user

representation.

User Embedding Layer. Currently, we use a simple user embedding layer to learn the high-level user representation based only on user id. We first map all users to a low-dimensional vector space. Each user u is described by a unique embedding vector $\mathbf{e}_u \in \mathbb{R}^d$. The embeddings of all users are implemented as a simple lookup table: $\mathbf{E}_u = [\mathbf{e}_1, ..., \mathbf{e}_m]$. These user embeddings are randomly initialized and will be updated in an end-to-end fashion during training the whole PDQN agent.

It is worth noting that a more complex user representation learning module can be designed and easily applied to PDQN. Any types of useful information that describes specific user can be exploited to learn the high-level user representation. For example, one can exploit demographics, user reviews, or social networks to learn user representation via proper deep neural networks in different recommendation domains. In this work, however, we only use the simplest form, an embedding layer, to implement and validate our core idea. On the other hand, since the embedding layer only requires user ids for inputs, the PDQN agent is applicable to any types of real-world recommender systems.

MLP$_U$ Layers. The user-specific action-value predicting module is a standard MLP with architecture $d \to \cdots \to 1$, denoted by MLP$_U$. After the embedding of user u, $\mathbf{e}_u \in \mathbb{R}^d$, is obtained, we employ MLP$_U$ to extract useful feature information from it and predict the user-specific action-value $Q_U(u)$:

$$\begin{aligned} \mathbf{c}_U^1 &\leftarrow \text{relu}(\mathbf{W}_U^1 \mathbf{e}_u + \mathbf{b}_U^1), \\ &\cdots \\ Q_U(u) &\leftarrow {\mathbf{w}_U^l}^\top \mathbf{c}_U^{l-1} + \text{b}_U^l, \end{aligned} \tag{6.5}$$

where $\cdot^\top$ is the transpose operation, $\mathbf{c}_U^i$, $\mathbf{W}_U^i$ (or $\mathbf{w}_U^i$) and $\mathbf{b}_U^i$ (or b_U^i) denote the outputs, trainable weights and biases of the i-th layer of MLP$_U$, respectively.

The General Action-value Function Q_G

The goal of Q_G is to predict the general action-value $Q_G(s_t, a)$ for each state-action pair (s_t, a), independent to specific users. Q_G consists of two basic modules: the state/action representation learning module that aims to learn high-level vector representations of state s_t and action a, and the general action-value predicting module that attempts to estimate the general action-value $Q_G(s_t, a)$ based on the concatenation of the learned high-level state and action representations. Here, we redefine the state s_t of user u as an item sequence, $\{a_0, ..., a_{t-1}\}$, denoting the items that user u has consumed before time step t[3].

Item Embedding Layer. In Q_G, we use a simple item embedding layer to learn some high-level item representations for the items in state s_t and action a, based only on item ids. We first map all items to a low-dimensional vector space. Each item i is described by a unique embedding vector $\mathbf{e}_i \in \mathbb{R}^d$. The embeddings of all items are implemented as a simple lookup table: $\mathbf{E}_i = [\mathbf{e}_1, ..., \mathbf{e}_n]$. Similarly, these item embeddings are randomly initialized and will be updated in an end-to-end fashion during training the whole PDQN agent. The learned item embedding of action a, $\mathbf{e}_a$, will be treated as the high-level action representation. The learned item embeddings of state s_t, $\{\mathbf{e}_{a_0}, ..., \mathbf{e}_{a_{t-1}}\}$, will be fed to the next layers for further processing.

GRU Layer. To model the sequential information in the state, as the exiting work did [158], we leverage a gated recurrent unit (GRU)[4] to further process the item embeddings of state s_t, $\{\mathbf{e}_{a_0}, ..., \mathbf{e}_{a_{t-1}}\}$, which can be simply denoted by $\{\mathbf{e}_0, ..., \mathbf{e}_{t-1}\}$ for notational convenience. The goal of this GRU layer is to transform $\{\mathbf{e}_0, ..., \mathbf{e}_{t-1}\}$

---

[3] We do not use the original state $s_t = \{(a_{t-k}, y_{ua_{t-k}}), ..., (a_{t-1}, y_{ua_{t-1}})\}$, since how to effectively exploit the user feedback information $\{y_{ua_0}, ..., y_{ua_{t-1}}\}$ to estimate action-values is also a challenging problem. Simple use (e.g., DEERS [158]) will heavily degenerate the agent's performance in our recommendation tasks. We leave the exploration to this problem in the future work.

[4] We chose GRU instead of long-short term memory (LSTM) and simple RNN because GRU has shown advantages over them in many recommendation tasks [44, 158].

to a sequence of hidden vectors $\{\mathbf{h}_0, ..., \mathbf{h}_{t-1}\}$, where $\mathbf{h}_j \in \mathbb{R}^d$ is abstractly computed as:

$$\mathbf{h}_j \leftarrow \mathrm{GRU}(\mathbf{h}_{j-1}, \mathbf{e}_j). \tag{6.6}$$

Attention Layer. Furthermore, we employ a self-attention mechanism [73] to capture the importance of different items in the state, and summarize a high-level state representation $\mathbf{h}_{s_t}$:

$$\mathbf{h}_{s_t} \leftarrow \sum\nolimits_{j=0}^{t-1} \beta_j \mathbf{h}_j, \tag{6.7}$$

where β_j is the attention score that indicates how much feature information of item a_j will be extracted, which is computed by:

$$\beta_j \leftarrow \frac{\exp\left(\mathbf{w}_{sa}^\top \tanh(\mathbf{W}_{sa}\mathbf{h}_j)\right)}{\sum_{l=0}^{t-1} \exp\left(\mathbf{w}_{sa}^\top \tanh(\mathbf{W}_{sa}\mathbf{h}_l)\right)}, \tag{6.8}$$

where $\mathbf{W}_{sa} \in \mathbb{R}^{d \times d}$ and $\mathbf{w}_{sa} \in \mathbb{R}^d$ are trainable weights in the self-attention. The attention layer is able to autonomously select more important features from the input sequence, and thus help the agent capture the user's dynamic preference at each time step.

MLP$_G$ Layers. The general action-value predicting module is a standard MLP with architecture $2d \rightarrow \cdots \rightarrow 1$, denoted by MLP$_G$. Once the high-level state representation $\mathbf{h}_{s_t} \in \mathbb{R}^d$ and the high-level action representation $\mathbf{e}_a \in \mathbb{R}^d$ are ready, we employ MLP$_G$ to fuse useful feature information from the concatenation of them and predict the general action-value $Q_G(s_t, a)$:

$$\begin{aligned} \mathbf{c}_G^1 &\leftarrow \mathrm{relu}(\mathbf{W}_G^1[\mathbf{h}_{s_t} \oplus \mathbf{e}_a] + \mathbf{b}_G^1), \\ &\cdots \\ Q_G(s_t, a) &\leftarrow {\mathbf{w}_G^l}^\top \mathbf{c}_G^{l-1} + \mathrm{b}_G^l, \end{aligned} \tag{6.9}$$

where $\cdot^\top$ is the transpose operation, $\mathbf{c}_G^i$, $\mathbf{W}_G^i$ (or $\mathbf{w}_G^i$) and $\mathbf{b}_G^i$ (or b_G^i) denote the outputs, trainable weights and biases of the i-th layer of MLP$_G$, respectively.

6.3.3 Personalized Q-learning

To train PDQN, i.e., the Q-network $Q(u, s, a; \theta)$, we adopt a Q-learning [145] based algorithm that minimizes the loss function:

$$\mathcal{L}(\theta) = \mathbb{E}_{u,s_t,a_t,r_{t+1},s_{t+1}} \left[(y - Q(u, s_t, a_t; \theta))^2 \right], \tag{6.10}$$

where $(u, s_t, a_t, r_{t+1}, s_{t+1})$ denotes a transition collected from user u's MDP_u, and θ denote all the trainable parameters of the Q-network. $y = r_{t+1} + \gamma \max_a Q(u, s_{t+1}, a; \theta^-)$ is the Q-learning target for current iteration, γ is the discount factor that balances the importance between future rewards and immediate rewards, and θ^- are the Q-network parameters from previous iteration, which are held fixed when performing optimization. In practice, instead of optimizing the full expectations in the above loss function, a more convenient way is to perform stochastic gradient descent (SGD) on a collected transition $(u, s_t, a_t, r_{t+1}, s_{t+1})$:

$$\theta \leftarrow \theta + \alpha \cdot (y - Q(u, s_t, a_t; \theta)) \cdot \nabla_\theta Q(u, s_t, a_t; \theta). \tag{6.11}$$

The entire personalized Q-learning algorithm is presented in Algorithm 6.1. To make the Q-network converge well, sufficient transitions that involve all possible states and actions are needed for Q-learning updates [119]. To this end, in each episode, we uniformly sample a user u from training set $\mathcal{U}_{train}$ as the current target user. User u's MDP_u will be used to interact with the agent and generate corresponding states and rewards. To ensure exploration, in each state s_t of MDP_u, the agent uses a ϵ-greedy policy that selects a greedy action $a_t = \arg\max_a Q(u, s_t, a; \theta)$ with probability $1 - \epsilon$ and a random action with probability ϵ. At the same time, the collected transition $(u, s_t, a_t, r_{t+1}, s_{t+1})$ is used to update the Q-network's parameters according to Equation 6.11. The learned agent can be used to make real-time recommendations for any testing user v. It only needs to interact with user v for an

Algorithm 6.1: Personalized Q-learning

Input: the user set $\mathcal{U}$, the training feedback data Y^{train}
Output: the learned Q-network $Q(u, s, a; \theta)$

1 **for** *episode* $= 1, ..., N$ **do**
2   Uniformly sample a target user u from $\mathcal{U}$
3   Initialize state s_0 as a pre-observed item sequence
4   **for** $t = 0, ..., T-1$ **do**
5   Recommend the ϵ-greedy item a_t w.r.t. $Q(u, s_t, a; \theta)$
6   Set reward $r_{t+1} = y_{ua_t}$ and state $s_{t+1} = s_t \cup \{a_t\}$
7   Update Q's weights θ according to **Equation 6.11**

episode, observe state s_t, predict the action-values $Q(v, s_t, a; \theta)$ for all actions, and recommend the greedy item $a_t = \arg\max_a Q(u, s_t, a; \theta)$, at each time step t.

Time Complexity. In the inner for-loop in Algorithm 6.1, the computation time is mainly taken in computing Q-values for all available items (line 5), and in updating Q-network (line 7). The cost of computing Q-values is $O(n|\theta|)$, where $|\theta|$ is the number of Q-network parameters and n is the number of items, and the cost of updating Q-network is $O(|\theta|)$. Therefore, the time complexity of training GCQN in worst case is $O(NTn|\theta|)$, where N is the number of episodes and T is the number of time steps in each episode.

6.3.4 Personalized Q-learning with Collaborative Regularizer

In order to make PDQN to approximate the optimal personalized action-value function, more effectively and efficiently, we propose a novel Q-learning algorithm that minimizes the loss function:

$$\mathcal{L}(\theta) = \mathbb{E}_{u,s_t,a_t,r_{t+1},s_{t+1}} \left[(y - Q(u, s_t, a_t; \theta))^2 + \lambda \cdot CR \right], \tag{6.12}$$

where $y = r_{t+1} + \gamma \max_a Q(u, s_{t+1}, a; \theta^-)$ is the Q-learning target of PDQN, λ is the regularization parameter, and CR is the *collaborative regularizer* defined as:

$$CR = \sum_{v \in N(u)} sim(u, v) \cdot \left(Q(v, s_t, a_t; \theta^-) - Q(u, s_t, a_t; \theta)\right)^2. \qquad (6.13)$$

Here, $N(u)$ denotes the set of collaborators of user u, i.e., the users who are most similar to user u according certain similarity metric. $sim(u, v) \in [0, 1]$ is the similarity between user u and his/her collaborator v. $Q(v, s_t, a_t; \theta^-)$ is the personalized action-value of triplet (v, s_t, a_t) predicted by PDQN at previous iteration, which indicates how good it is to take action a_t in state s_t for user v, over the long run.

The intuition behind the CR regularizer is straightforward, that is, the personalized action-values between two similar users are inclined to be close to each other. By leveraging the CR regularizer, PDQN is able to model the collaborative relationships between different users, more effectively and efficiently. As a result, a better personalized action-value function $Q(u, s, a)$ can be estimated by PDQN, which leads to a better personalized policy $\pi(u, s)$.

The new personalized Q-learning algorithm is presented in Algorithm 6.2. To reduce the computational complexity, when optimizing the loss function in Equation 6.12, we uniformly sample a single collaborator v from $N(u)$ to compute the CR regularizer, instead of the whole set. Given a collected transition $(u, s_t, a_t, r_{t+1}, s_{t+1})$, the new SGD update is:

$$\begin{aligned} \theta \leftarrow \theta + \alpha \cdot [&(y - Q(u, s_t, a_t; \theta)) \\ &+ \lambda \cdot sim(u, v) \cdot (Q(v, s_t, a_t; \theta^-) - Q(u, s_t, a_t; \theta)) \\ &] \cdot \nabla_\theta Q(u, s_t, a_t; \theta). \end{aligned} \qquad (6.14)$$

We refer to the PDQN agent trained with the new personalized Q-learning algorithm as PDQN-cr.

Algorithm 6.2: Personalized Q-learning with Collaborative Regularizer

Input: the user set $\mathcal{U}$, the training feedback data Y^{train}, the collaborator set $N(u), \forall u \in \mathcal{U}$, the similarity $sim(u, v), \forall u, v \in \mathcal{U}$

Output: the learned Q-network $Q(u, s, a; \theta)$

1 **for** *episode* $= 1, ..., N$ **do**
2   Uniformly sample a target user u from $\mathcal{U}$
3   Initialize state s_0 as a pre-observed item sequence
4   **for** $t = 0, ..., T-1$ **do**
5   Recommend the ϵ-greedy item a_t w.r.t. $Q(u, s_t, a; \theta)$
6   Set reward $r_{t+1} = y_{ua_t}$ and state $s_{t+1} = s_t \cup \{a_t\}$
7   Uniformly sample a collaborator v from $N(u)$
8   Update Q's weights θ according to **Equation 6.14**

Detailed Designs in CR

The similarity metric is nontrivial to the performance of the CR regularizer. In this work, we use the cosine similarity to compute $sim(u, v)$ between user u and user v based on some feature vectors of them:

$$sim(u, v) = \frac{\mathbf{p}_u{}^\top \mathbf{p}_v}{\|\mathbf{p}_u\|_2 \cdot \|\mathbf{p}_v\|_2}, \tag{6.15}$$

where $\|\cdot\|_2$ denotes the L^2 norm of vectors, and $\mathbf{p}_u$ ($\mathbf{p}_v$) denotes the low-dimensional feature vector of user u (v).

The low-dimensional feature vectors of all users are trained by a matrix factorization (MF) model based on the training feedback data Y^{train}. The MF loss function to be minimized is defined as:

$$\mathcal{L} = \sum\nolimits_{u,i} (\mathbf{p}_u{}^\top \mathbf{q}_i - y_{ui})^2 + \lambda_{mf} (\sum\nolimits_u \|\mathbf{p}_u\|_2^2 + \sum\nolimits_i \|\mathbf{q}_i\|_2^2), \tag{6.16}$$

where y_{ui} is either an observed positive feedback ($y_{ui} = 1$) in Y^{train} or a sampled negative feedback ($y_{ui} = 0$), λ_{mf} is the regularization parameter, and $\mathbf{p}_u \in \mathbb{R}^d$ and $\mathbf{q}_i \in \mathbb{R}^d$ denote the feature vectors of user u and item i, respectively. To optimize the above loss function, in each iteration, we traverse all the observed positive feedbacks in Y^{train}. For each traversed positive feedback $y_{ui} = 1$, we uniformly sample a

negative feedback $y_{uj} = 0$ from the unobserved feedbacks of user u. Then, we perform two SGD updates to the corresponding feature vectors based on y_{ui} and y_{uj}, respectively. The update rule for y_{ui} is given by:

$$\mathbf{p}_u \leftarrow \mathbf{p}_u - \alpha \left[(\mathbf{p}_u{}^\top \mathbf{q}_i - y_{ui}) \mathbf{q}_i + \lambda_{mf} \mathbf{p}_u \right]$$
$$\mathbf{q}_i \leftarrow \mathbf{q}_i - \alpha \left[(\mathbf{p}_u{}^\top \mathbf{q}_i - y_{ui}) \mathbf{p}_u + \lambda_{mf} \mathbf{q}_i \right], \tag{6.17}$$

The time complexity of training the feature vectors is $O(cd|\Omega|)$, where c is the number of iterations, d is the dimensionality of latent space, and $|\Omega|$ is the number of observed positive feedbacks in Y^{train}. As both c and d are usually small constants (we set $c = 20$ and $d = 64$ in our experiments) in practice, the cost is quite low, which is linear to the size of observed feedback data. Note that we tested $c \in \{10, 20, 30\}$ in our experiments, but the results did not show significant differences.

Once the similarities between users are computed, we select k most similar collaborators to form the set $N(u)$ for each user u. It is worth noting that the CR regularizer is generic, which does not limited to the specific similarity metric described above. More complex ways can be used to compute the similarities between users, in order to further improve the performance of CR. For example, one may exploit an additional social network of users and compute their similarities according the social relations between them.

6.4 Experiments

We conducted extensive experiments to validate the proposed PDQN methods. In this section, we describe our experiments and show the results and analysis.

Table 6.1: The statistics of the Amazon datasets.

Statistics	Beauty	Music	Phone	Video
#users	2,826	1,835	1,141	273
#items	42,042	41,488	18,797	5,076
#positive feedbacks	97,860	75,932	34,653	9,407
#density	0.08%	0.10%	0.16%	0.68%

6.4.1 Experimental Setup

Datasets. We employ four datasets with different sizes and domains from the Amazon product data repository[5] [41]: "Beauty" (**Beauty**), "Digital Music" (**Music**), "Cell Phones and Accessories" (**Phone**), and "Amazon Instant Video" (**Video**). All datasets contain explicit user ratings on items. Since we focus on implicit-feedback recommendations, we follow the common practice to convert all ratings to 1 to represent the implicit positive feedbacks. To ensure there is enough data to conduct experiments for interactive recommendations, we remove the users with fewer than 20 positive feedbacks for each dataset. A summary of characteristics of the obtained datasets is given in Table 6.1.

Evaluation Methodology. We evaluate the recommendation agent on the following T-step interactive recommendation task. Suppose target user u has n_u observed positive feedbacks in the dataset. Since not all unobserved items are truly negative, we randomly select the same number of unobserved (u, i) pairs of user u as the n_u sampled negative feedbacks ($y_{ui} = 0$). In each episode of the interactive recommendation process, the agent is forced to pick items from the available item set that consists of the n_u sampled negative items and the n_u observed positive items. An episode ends at time step T if and only if the agent has recommended all the n_u positive items to user u before time step T. This implies that T varies with respect to different users and different agents. For an ideal agent, $T = n_u$.

[5]http://jmcauley.ucsd.edu/data/amazon/

For each dataset, we split the observed feedback data Y into Y_{train} and Y_{test} with the follow procedure. We first randomly choose 10% users as testing user set $\mathcal{U}_{test}$, and the remaining 90% users as the training user set $\mathcal{U}_{train}$. All positive feedbacks of $\mathcal{U}_{train}$ are moved from Y to Y_{train}. For each testing user $v \in \mathcal{U}_{test}$, the entire n_v positive feedbacks in Y are divided equally into Y_{train} and Y_{test}. The final obtained Y_{train} and Y_{test} are used to train and test the RL agents, respectively. Note that for each testing user, the sampled negative item set for training and that for testing are forced to be disjoint. We conduct each experiment on 5 data splits obtained with different random seeds.

The evaluation metric we used is the average reward (AvgReward) the agent received in the T-step episode for each testing user v. Recall that the reward of a positive (negative) item is 1 (-1). Thus, *AvgReward* is defined as:

$$\text{AvgReward} = \frac{n_v - (T - n_v)}{T} = \frac{2n_v - T}{T}, \tag{6.18}$$

where n_v denotes the number of recommended positive items (equal to the number of actual positive items in Y_{test}), and $T - n_v$ denotes the the number of recommended negative items. Clearly, AvgReward $\in [0, 1]$. The final value of AvgReward is obtained by taking its average over all testing users and 5 data splits.

Baselines. To comparatively evaluate our proposed methods, **PDQN** and **PDQN-cr**, we carefully choose a number of representative **RL-based baselines** from the literature:

- **DEERS** [158]. This is a DQN-based recommendation method, which utilizes GRU to learn state representations from positive- and negative-feedback item sequences.

- **UDQN** [63]. This is a DQN-based recommendation method, which utilizes matrix factorization to learn latent states.

- **DQN** [90]. This the original DQN method designed for playing Atari. We adapt it to our task by using the same architecture of the general action-value function Q_G in PDQN.

- **DQN-u**. This method extends the above DQN by taking the high-level user representation in PDQN as an additional part of state to feed the general action-value predicting module.

All of the above baselines are Q-network agents. The major difference between them and our PDQN is that they fail to model the factors of personalization and collaboration of users, or only model the factors implicitly and indirectly via the state representations. For fair comparisons, we train these agents using the same Q-learning algorithm of PDQN in Algorithm 6.1. This enables us to focus on validating the efficacy of our personalized action-value predicting network architecture, i.e., the PDQN model.

Besides, we compare several **non-RL baselines** for reference:

- **Random** picks items randomly. It can be seen as an indicator that reveals the difficulty of the task itself.

- **Popular** picks items with most positive feedbacks. It is a simple but strong baseline in many recommendation tasks.

- **SVD** [59] is a matrix factorization model that uses the inner product of user- and item-specific latent feature vectors to predict user-item relevance scores[6].

- **LinUCB** [69] is a contextual bandit method for news recommendation. We adapt it to our task by concatenating the SVD-based user and item feature

[6] We have tried NCF [42] that utilizes MLP to replace the inner product in SVD. However, this more complex model failed to produce better results than SVD in a reasonable amount of training time. Since both SVD and NCF are essentially the same supervised learning based models, we only report the results of SVD here.

Table 6.2: The parameter settings of PDQN-cr.

Parameter	**Beauty**	**Music**	**Phone**	**Video**
CR regularization λ	0.1	0.3	0.5	0.1
neighborhood size N	5	10	10	5

vectors as its context vector.

Parameter Settings. We implement the compared RL-based methods in PyTorch. Since all of them are trained with the same Q-learning algorithm, we use the DQN as a base model to tune some common hyperparameters using grid search, which will be fixed to all methods. This setting is reasonable as it enables us to make a fair comparison on the particular action-value predictors in different methods. More specifically, we adopt the Adam optimizer to update Q-networks. All parameters in Adam are set to the default values in PyTorch, with one exception of learning rate $\alpha = 0.0001$. Other shared hyperparameters are set as follows: the embedding size $d = 64$, the discount factor $\gamma = 0.5$, the ϵ-greedy parameter $\epsilon = 0.1$, the number of training episodes $N = 4\times10^4$, and the architecture of the MLP_G $128 \rightarrow 64 \rightarrow 32 \rightarrow 1$. For PDQN and PDQN-cr, the architecture of the MLP_U is $64 \rightarrow 32 \rightarrow 16 \rightarrow 1$. For DEERS and UDQN, we adopt the same architecture described in the original papers. For SVD, we set the latent dimension as 64, the regularization parameter as 0.001, and the learning rate as 0.01. For PDQN-cr, we use the same parameters of SVD to train the user and item embeddings. Besides, the settings of the neighborhood size N and the CR regularization parameter λ are given in Table 6.2.

6.4.2 Comparison Results

The comparison results of all methods, in terms of the AvgReward received in the interactive recommendation process, are reported in Table 6.3. The best performing method is highlighted in bold font (always PDQN-cr). The best performing baseline

Table 6.3: Performance comparison of all methods, in terms of the AvgReward. The best performing method is highlighted in bold font (always PDQN-cr). The best performing baseline is marked with "*" (always DQN).

Type	Method	Beauty	Music	Phone	Video
Non-RL	Random	0.019	0.023	0.017	0.006
	Popular	0.077	0.115	0.008	0.009
	SVD	0.094	0.208	0.067	0.083
	LinUCB	0.094	0.198	0.048	0.064
RL	DEERS	0.124	0.229	0.059	0.072
	UDQN	0.151	0.243	0.093	0.110
	DQN	0.179*	0.298*	0.105*	0.152*
	DQN-u	0.151	0.273	0.104	0.141
PRL	PDQN	0.183	0.324	0.127	0.172
	PDQN-cr	**0.200**	**0.326**	**0.136**	**0.198**

is marked with "*" (always DQN).

- Our PDQN shows remarkable advantages over the baselines on all datasets. Its relative improvements over the best performing baseline DQN are about 2.5%, 8.6%, 20.5% and 13.8% on the Beauty, Music, Phone and Video datasets, respectively[7]. *This clearly demonstrates that the proposed personalized Q-network architecture is able to learn personalized recommendation policies more effectively than the existing deep RL agents.*

- PDQN-cr further improves the performance of PDQN by approximately 9.3%, 0.6%, 7.1% and 15.1% on the Beauty, Music, Phone and Video datasets, respectively. *This highly indicates that the proposed collaborative Q-learning objective, i.e., the loss function with the collaborative regularizer CR, is very helpful to the learning of effective personalized recommendation policies.*

[7]Note that in the famous Netflix Prize, the winner only improves the baseline 10%!

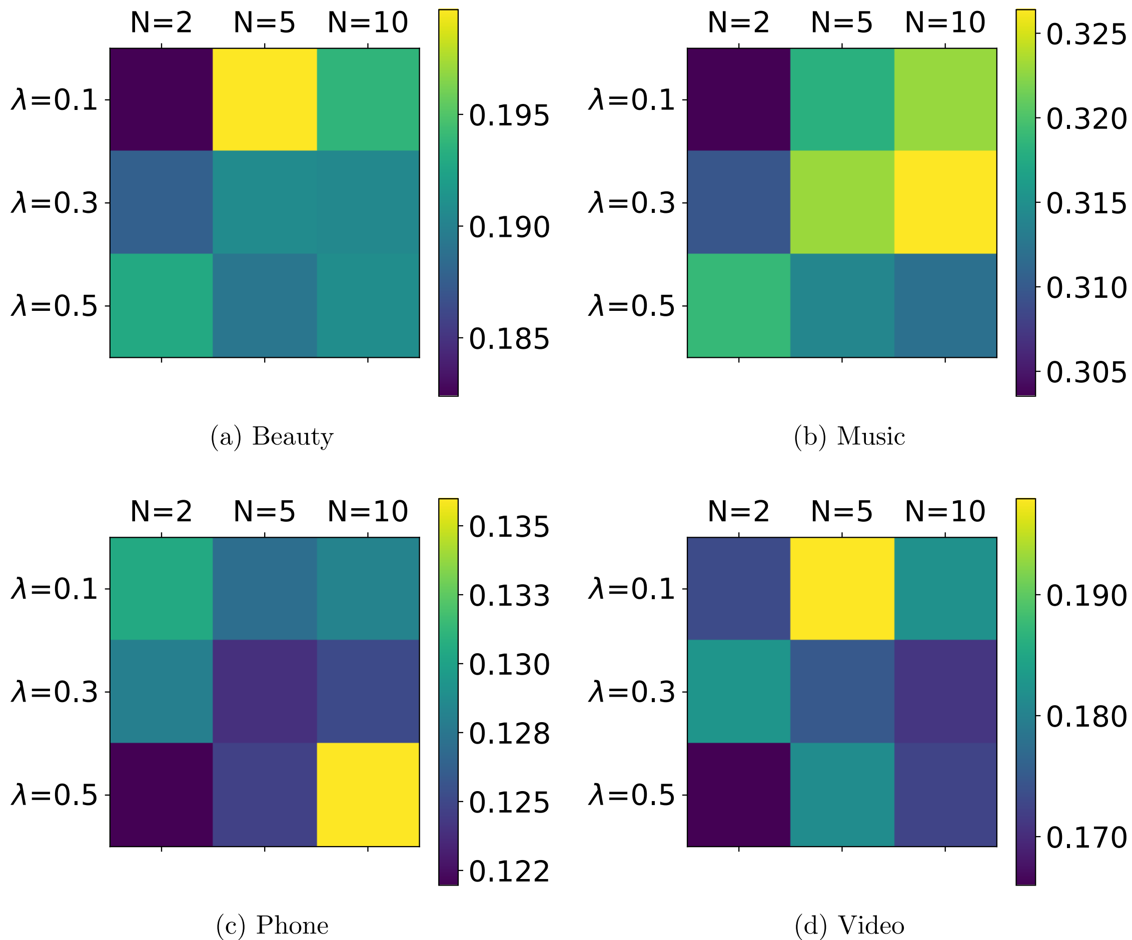


Figure 6.4: The impact of the CR regularization parameter λ and the size of collaborative neighborhood N on the performance of PDQN-cr on each dataset.

6.4.3 Parameter Analysis

We analyze the impact of two important hyperparameters on the performance of PDQN-cr: the CR regularization parameter λ and the size of collaborative neighborhood N. To this end, we vary $\lambda \in \{0.1, 0.3, 0.5\}$ and $N \in \{2, 5, 10\}$ to check how PDQN-cr behaves on each dataset. The comparison results is shown in Figure 6.4. We find that the combination of $\lambda = 0.1, N = 5$ achieves best performance on the Beauty and Video datasets, $\lambda = 0.3, N = 10$ achieves best performance on the Music dataset, and $\lambda = 0.5, N = 10$ achieves best performance on the Phone dataset.

6.5 Conclusions and Discussions

Existing RL-based recommendation agents are significantly limited by a common weakness that the key factors of personalization and collaboration are not modeled or only modeled in states and actions in an implicit way, which are not able to learn a fully personalized recommendation policy. In this work, we propose a novel personalized reinforcement learning (PRL) paradigm, a more explicit and effective approach to modeling personalization and collaboration in RL-based recommendation agents. To implement the idea of PRL, we develop an end-to-end value-based agent, term Personalized Deep Q-network (PDQN), which is able to learn a fully personalized policy that makes recommendations based on both the user-specific information of a particular user and the general information of state shared by all users. Furthermore, we propose a novel collaborative Q-learning objective that contains a particular collaborative regularizer in order to further model collaboration, which is able to make PDQN to approximate the optimal personalized policy more effectively than conventional Q-learning objective. We validate PDQN and the collaborative regularizer by conducting solid experiments on four real-world datasets.

Our approaches can be extended from the following directions. First of all, the network architecture of PDQN can be further improved by using some more complex user representation modules and state/action representation modules. Second, the collaborative regularizer can also be extended with other similarity metrics and user embedding methods or data sources. Last but not least, the idea of PRL and the technical designs in PDQN can be readily transferred to policy-based deep RL agents. We will explore these problems in the future work.

Chapter 7

Conclusions and Suggestions for Future Research

Recommender system is a powerful tool for information filtering, which has been widely and successfully applied on the web. The core of a recommender system is the recommendation algorithm/method that determines what items should be recommended to users. Over the past two decades, researchers from different areas have proposed a wide variety of recommendation algorithms and achieved huge successes. However, the majority of existing recommendation algorithms are essentially *supervised learning* (SL) based approaches that only aim to learn static, passive, and shortsighted predictive models for the single-step recommendation problem, but cannot provide satisfactory solutions the more practical multi-step interactive recommendation problem.

To address such issue, a potential way is to leverage the *reinforcement learning* (RL) paradigm to build an RL-based recommendation agent. A key feature of RL-based agents is that they are able to model the user's dynamic long-term preferences towards future, and aim to learn a proactive and farsighted recommendation policy that optimizes the cumulative rewards received in the multi-step interactive recommendation process. Unfortunately, the existing RL-based recommendation agents fail to effectively model the crucial factors of personalization and collaboration, which

cannot provide users with truly personalized recommendations that optimize the overall profit of the recommender system.

In this thesis, we develop four novel and effective value-based deep RL recommendation agents, by effectively modeling personalization and collaboration into one or more of the three modules: *the state/action representation module*, *the action-value prediction module*, and *the Q-learning module*, in different ways. Chapter 3 presents User-specific Deep Q-network (UDQN), a two-stage pipeline agent that first constructs latent vector representations of states and actions using matrix factorization (MF) and then estimates action-values based on those latent representations using Q-learning. Chapter 4 describes Graph Convolutional Q-network (GCQN), an end-to-end agent that directly estimates action-values based on the input of graph-structured representations of states and actions by successfully processing them using graph convolutional networks (GCNs). In the proposed UDQN and GCQN agents, the factors of personalization and collaboration are effectively modeled via the MF-based and the GCN-based state/action representation modules, respectively. Chapter 5 introduces Social Attentive Deep Q-network (SADQN), a two-stage pipeline agent that estimates action-values based on both personal preferences and social neighbors' preferences by using the personal action-value function and the social action-value function, respectively. On the basis of UDQN, the proposed SADQN agent further models personalization and collaboration explicitly into the action-value prediction module by successfully leveraging social attention to capture the social influence between target user and his/her social neighbors in the social network. Chapter 6 presents Personalized Deep Q-network (PDQN), an end-to-end agent that estimates action-values based on both user-specific information of the user and general information of the state by using the user-specific action-value function and the general action-value function, respectively. Unlike the above three agents, the proposed PDQN agent utilizes a brand-new personalized Q-network architecture (i.e.,

the combination of the first two modules) to model personalization and collaboration, which is able to produce a fully personalized recommendation policy. Moreover, the collaboration is further modeled by PDQN in a collaborative Q-learning module.

7.1 Summary of Contributions

The following sections summarize the contributions of each chapter.

7.1.1 UDQN: Modeling Personalization and Collaboration via MF-based Representation Module

- This work is the first attempt to explore the combination of CF and RL for recommendation problems. We seamlessly integrates the ideas of CF and RL by designing an MF-based state/action representation module for the RL-based recommendation agent.

- The proposed MF-based representation module successfully models the factors of personalization and collaboration by mapping all users into a shared latent feature space, and produces informative MF-based vector representations of states and actions for policy learning.

- We develop a two-stage pipeline agent, named UDQN, which is able to learn an effective recommendation policy based on the constructed MF-based state/action representations through Q-learning.

- We verify the efficacy and capability of UDQN with comprehensive experimental results and analysis on real-world datasets, in terms of both explicit-feedback and implicit-feedback recommendation tasks. We find that the MF-based representation module helps a lot in learning the RL-based recommendation policy. Moreover, the learned policy is highly consistent with the quality

of the specific MF model adopted in the representation module. A better MF model will lead to a better policy.

- Our approach is generic and flexible, which can be readily extended with more complex MF models and Q-network architectures, and applied to other related recommendation scenarios and datasets.

7.1.2 GCQN: Modeling Personalization and Collaboration via GCN-based Representation Module

- This is the first work that incorporates GCN to design an RL-based recommendation agent. We propose a GCN-based state/action representation module that is able to transform the graph-structured representations to meaningful vector representations.

- The proposed GCN-based representation module successfully models the factors of personalization and collaboration by aggregating valuable features from target user's local neighborhood in the user-item bipartite graph, in terms of the feature propagation on the "user-item-user" paths.

- We develop an end-to-end agent, termed GCQN, which is able to directly approximate the optimal action-value function (corresponding an optimal recommendation policy) based on the input of graph-structured representations.

- We conduct extensive experiments on three real-world datasets. The results demonstrate that: (1) the proposed GCN-based representation module helps a lot in learning farsighted recommendation policies, and (2) GCQN is effective, robust and efficient, which achieves significant performance gains over state-of-the-art baselines with reasonable computation cost.

- Our approach is a generic framework for graph-structure reinforcement learning, which can be readily extended with more advanced techniques in GCN designs (e.g., the fastGCN).

7.1.3 SADQN: Improving Personalization and Collaboration via Social Attention

- We make the first attempt to improve the performance of RL-based recommendation agents by effectively utilizing available social networks of users.

- We develop a two-stage pipeline agent, termed SADQN, which further models the factors of personalization and collaboration, on the basis of UDQN, by estimating action-values based on the combination of a personal action-value function and a social action-value function. The personal and social action-value functions aim to estimate action-values based on the preferences of individual users and social friends, respectively.

- In particular, the social action-value function successfully models the collaborations between target user and his/her social neighbors by leveraging social attention to capture the social influence between them.

- Further, we develop an enhanced variant of SADQN, termed SADQN++, to model the complicated and diverse trade-offs between personal preferences and social influence for all involved users, making the agent more powerful and flexible in learning optimal policies.

- The solid experimental results on real-world datasets demonstrate that the proposed SADQNs remarkably outperform the state-of-the-art agents, especially in the cold-start recommendation scenario. We also find that, the better modeling of social influence (i.e., the collaboration between target user and social friends), the better recommendation performance the agent will achieve.

7.1.4 PDQN: Integrating Personalized Network Architecture with Collaborative Learning Objective

- We propose a novel end-to-end agent, named PDQN, which is able to learn a fully personalized recommendation policy that depends on both the user-specific information of individual user and the general information of state shared by all users.

- Unlike the UDQN and GCQN agents that model personalization and collaboration implicitly into state/action representations, PDQN explicitly models them into the whole Q-network architecture, by estimating personalized action-values based on the combination of a user-specific action-value function and a general action-value function. Besides, unlike the SADQN agent, PDQN does not rely on additional social information, which can be applied to more recommendation scenarios and datasets.

- We design a novel collaborative Q-learning objective to further model collaboration between similar neighbors, which is able to make PDQN to approximate the optimal personalized policy more effectively.

- We conduct solid experiments on several real-world datasets to validate our methods. The results sufficiently verify the efficacy of both the personalized Q-network architecture and the collaborative Q-learning objective.

7.2 Future Work

For future work, we point out several potential directions in the following.

Each of the proposed agents could be further enhanced by employing more advanced techniques in its core module. A number of extensions could be made in this direction. For UDQN, the standard MF model in the MF-based representation

module can be replaced by more complex models such as SVD++ [57] and LLORMA [61]. For GCQN, the GCN design in the CGN-based representation module can be replaced with more recent techniques such as fastGCN [16] and Geom-GCN [95]. For SADQN, the social attention layer can be enhanced by using more complex attention mechanisms such as multi-head attention [127]. For PDQN, the simple user embedding layer can be replaced by more complex user representation methods, and the collaborative regularizer can also be extended with other similarity computation approaches. For all agents, the general action-value prediction module can be replaced by the dueling network architectures [144], in order to improve policy learning in those cases where many similar-valued actions exist.

The ideas of modeling personalization and collaboration in the proposed agents are largely complementary to each other, which can be properly integrated to further improve the agent's performance. In fact, the proposed SADQN agent integrates the ideas of both MF-based representation and social attention. In the future, we will explore the following extensions. First, the GCN-based state/action representation module in GCQN can be incorporated into SADQN and PDQN to further improve their performance. Second, the personalized network architecture in PDQN can be applied to SADQN, in order to better estimate the action-values in social context. Moreover, the collaborative learning objective in PDQN can be extended by leveraging the available social network of users.

Currently, we only implement our ideas of modeling personalization and collaboration on the basis of value-based deep RL agents. Are the proposed ideas also beneficial to policy-based deep RL agents such as DDPG [71]? We will explore this problem in future work. We are also interested in how to design more effective deep RL agents by leveraging the available knowledge graph on the item side.

Bibliography

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *TKDE*, 17(6):734–749, 2005.

[2] N. Alon, B. Awerbuch, Y. Azar, and B. Patt-Shamir. Tell me who i am: an interactive recommendation system. *Theory of Computing systems*, 45(2):261–279, 2009.

[3] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[4] E. Bakshy, I. Rosenn, C. Marlow, and L. Adamic. The role of social networks in information diffusion. In *WWW*, pages 519–528. ACM, 2012.

[5] S. Balakrishnan and S. Chopra. Collaborative ranking. In *WSDM*, pages 143–152. ACM, 2012.

[6] R. Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[7] R. v. d. Berg, T. N. Kipf, and M. Welling. Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263*, 2017.

[8] D. A. Berry and B. Fristedt. Bandit problems: sequential allocation of experiments. *Monographs on statistics and applied probability*, 5:71–87, 1985.

[9] R. M. Bond, C. J. Fariss, J. J. Jones, A. D. Kramer, C. Marlow, J. E. Settle, and J. H. Fowler. A 61-million-person experiment in social influence and political mobilization. *Nature*, 489(7415):295, 2012.

[10] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *UAI*, pages 43–52. Morgan Kaufmann Publishers Inc., 1998.

[11] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. In *ICLR*, 2014.

[12] I. Cantador, P. L. Brusilovsky, and T. Kuflik. *HetRec2011*. ACM, 2011.

[13] A. J. B. Chaney, D. M. Blei, and T. Eliassi-Rad. A probabilistic model for using social networks in personalized item recommendation. In *RecSys*, pages 43–50. ACM, 2015.

[14] C. Chen, M. Zhang, Y. Liu, and S. Ma. Social attentional memory network: Modeling aspect-and friend-level differences in recommendation. In *WSDM*, pages 177–185. ACM, 2019.

[15] H. Chen, X. Dai, H. Cai, W. Zhang, X. Wang, R. Tang, Y. Zhang, and Y. Yu. Large-scale interactive recommendation with tree-structured policy gradient. In *AAAI*, volume 33, pages 3312–3320, 2019.

[16] J. Chen, T. Ma, and C. Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. In *ICLR*, 2018.

[17] J. Chen, C. Wang, S. Zhou, Q. Shi, Y. Feng, and C. Chen. Samwalker: Social recommendation with informative sampling strategy. In *WWW*, pages 228–239. ACM, 2019.

[18] M. Chen, A. Beutel, P. Covington, S. Jain, F. Belletti, and E. H. Chi. Top-k off-policy correction for a reinforce recommender system. In *WSDM*, pages 456–464. ACM, 2019.

[19] S. Chen, Y. Yu, Q. Da, J. Tan, H. Huang, and H. Tang. Stabilizing reinforcement learning in dynamic environment with application to online recommendation. In *SIGKDD*, pages 1187–1196. ACM, 2018.

[20] X. Chen, S. Li, H. Li, S. Jiang, Y. Qi, and L. Song. Generative adversarial user model for reinforcement learning based recommendation system. In *ICML*, pages 1052–1061, 2019.

[21] H. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10. ACM, 2016.

[22] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[23] S. Choi, H. Ha, U. Hwang, C. Kim, J. Ha, and S. Yoon. Reinforcement learning based recommender system using biclustering technique. *arXiv preprint arXiv:1801.05532*, 2018.

[24] K. Christakopoulou and A. Banerjee. Collaborative ranking with a push at the top. In *WWW*, pages 205–215. ACM, 2015.

[25] K. Christakopoulou, F. Radlinski, and K. Hofmann. Towards conversational recommender systems. In *SIGKDD*, pages 815–824. ACM, 2016.

[26] P. Cremonesi, Y. Koren, and R. Turrin. Performance of recommender algorithms on top-n recommendation tasks. In *RecSys*, pages 39–46. ACM, 2010.

[27] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, pages 3844–3852, 2016.

[28] M. Deshpande and G. Karypis. Item-based top-n recommendation algorithms. *TOIS*, 22(1):143–177, 2004.

[29] M. A. Domingues, F. Gouyon, A. M. Jorge, J. P. Leal, J. Vinagre, L. Lemos, and M. Sordo. Combining usage and content in an online recommendation system for music in the long tail. *IJMIR*, 2(1):3–13, 2013.

[30] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*, pages 2224–2232, 2015.

[31] T. Ebesu, B. Shen, and Y. Fang. Collaborative memory network for recommendation systems. In *SIGIR*, pages 515–524. ACM, 2018.

[32] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin. Graph neural networks for social recommendation. pages 417–426, 2019.

[33] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, et al. Noisy networks for exploration. In *ICLR*, 2018.

[34] X. Geng, H. Zhang, J. Bian, and T. Chua. Learning image and user features for recommendation in social networks. In *ICCV*, pages 4274–4282, 2015.

[35] J. Gittins, K. Glazebrook, and R. Weber. *Multi-armed bandit allocation indices.* John Wiley & Sons, 2011.

[36] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[37] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He. Deepfm: a factorization-machine based neural network for ctr prediction. In *IJCAI*, pages 1725–1731. AAAI Press, 2017.

[38] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017.

[39] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *TIIS*, 5(4):19, 2016.

[40] Hado V Hasselt. Double q-learning. In *NIPS*, pages 2613–2621, 2010.

[41] R. He and J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *WWW*, pages 507–517, 2016.

[42] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T. Chua. Neural collaborative filtering. In *WWW*, pages 173–182. IW3C2, 2017.

[43] X. He, H. Zhang, M. Kan, and T. Chua. Fast matrix factorization for online recommendation with implicit feedback. In *SIGIR*, pages 549–558. ACM, 2016.

[44] B. Hidasi, A. Karatzoglou, L. Baltrunas, and D. Tikk. Session-based recommendations with recurrent neural networks. In *ICLR*, 2016.

[45] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[46] B. Hu, C. Shi, and J. Liu. Playlist recommendation based on reinforcement learning. In *ICIS*, pages 172–182. Springer, 2017.

[47] J. Hu and P. Li. Collaborative multi-objective ranking. In *CIKM*, pages 1363–1372. ACM, 2018.

[48] Y. Hu, Q. Da, A. Zeng, Y. Yu, and Y. Xu. Reinforcement learning to rank in e-commerce search engine: Formalization, analysis, and application. In *SIGKDD*, pages 368–377. ACM, 2018.

[49] Y. Hu, Y. Koren, and C. Volinsky. Collaborative filtering for implicit feedback datasets. In *ICDM*, pages 263–272. IEEE, 2008.

[50] J. Huang, W. X. Zhao, H. Dou, J. Wen, and E. Y. Chang. Improving sequential recommendation with knowledge-enhanced memory networks. In *SIGIR*, pages 505–514, 2018.

[51] S. Huang, S. Wang, T. Liu, J. Ma, Z. Chen, and J. Veijalainen. Listwise collaborative filtering. In *SIGIR*, pages 343–352. ACM, 2015.

[52] M. Jamali and M. Ester. Trustwalker: a random walk model for combining trust-based and item-based recommendation. In *SIGKDD*, pages 397–406. ACM, 2009.

[53] M. Jamali and M. Ester. A matrix factorization technique with trust propagation for recommendation in social networks. In *RecSys*, pages 135–142. ACM, 2010.

[54] H. Kautz, B. Selman, and M. Shah. Referral web: combining social networks and collaborative filtering. *Communications of the ACM*, 40(3):63–65, 1997.

[55] J. Kawale, H. H. Bui, B. Kveton, L. Tran-Thanh, and S. Chawla. Efficient thompson sampling for online matrix-factorization recommendation. In *NIPS*, pages 1297–1305, 2015.

[56] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *ICLR*, 2017.

[57] Y. Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *SIGKDD*, pages 426–434. ACM, 2008.

[58] Y. Koren and R. Bell. Advances in collaborative filtering. In *Recommender systems handbook*, pages 77–118. Springer, 2015.

[59] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[60] J. Lee, S. Bengio, S. Kim, G. Lebanon, and Y. Singer. Local collaborative ranking. In *WWW*, pages 85–96. ACM, 2014.

[61] J. Lee, S. Kim, G. Lebanon, and Y. Singer. Local low-rank matrix approximation. In *ICML*, pages 82–90, 2013.

[62] Y. Lei and W. Li. Personalized deep q-network for interactive recommendation. *TOIS (Under review)*.

[63] Y. Lei and W. Li. Interactive recommendation with user-specific deep reinforcement learning. *TKDD*, 13(6):1–15, 2019.

[64] Y. Lei and W. Li. When collaborative filtering meets reinforcement learning. *arXiv preprint arXiv:1902.00715*, 2019.

[65] Y. Lei, W. Li, Z. Lu, and M. Zhao. Alternating pointwise-pairwise learning for personalized item ranking. In *CIKM*, pages 2155–2158. ACM, 2017.

[66] Y. Lei, H. Pei, H. Yan, and W. Li. Reinforcement learning based recommendation with graph convolutional q-network. In *SIGIR (Under review)*.

[67] Y. Lei, Z. Wang, W. Li, and H. Pei. Social attentive deep q-network for recommendation. In *SIGIR*, pages 1189–1192. ACM, 2019.

[68] Y. Lei, Z. Wang, W. Li, H. Pei, and Q. Dai. Social attentive deep q-networks for recommender systems. *TKDE (Major Revision)*.

[69] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *WWW*, pages 661–670. ACM, 2010.

[70] D. Liang, R. G. Krishnan, M. D. Hoffman, and T. Jebara. Variational autoencoders for collaborative filtering. In *WWW*, pages 689–698. International World Wide Web Conferences Steering Committee, 2018.

[71] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. In *ICLR*, 2016.

[72] L. Lin. Reinforcement learning for robots using neural networks. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1993.

[73] Z. Lin, M. Feng, C. N. d. Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio. A structured self-attentive sentence embedding. In *ICLR*, 2017.

[74] C. Liu, C. Zhou, J. Wu, Y. Hu, and L. Guo. Social recommendation with an essential preference space. In *AAAI*, 2018.

[75] F. Liu, H. Guo, X. Li, R. Tang, Y. Ye, and X. He. End-to-end deep reinforcement learning based recommendation with supervised embedding. In *WSDM*, pages 384–392, 2020.

[76] F. Liu, R. Tang, X. Li, W. Zhang, Y. Ye, H. Chen, H. Guo, and Y. Zhang. Deep reinforcement learning based recommendation with explicit user-item interactions modeling. *arXiv preprint arXiv:1810.12027*, 2018.

[77] Tie-Yan Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[78] Y. Liu, Y. Zhang, Q. Wu, C. Miao, L. Cui, B. Zhao, Y. Zhao, and L. Guan. Diversity-promoting deep reinforcement learning for interactive recommendation. *arXiv preprint arXiv:1903.07826*, 2019.

[79] M. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[80] H. Ma, I. King, and M. R. Lyu. Learning to recommend with social trust ensemble. In *SIGIR*, pages 203–210. ACM, 2009.

[81] H. Ma, I. King, and M. R. Lyu. Learning to recommend with explicit and implicit social relations. *TIST*, 2(3):29, 2011.

[82] H. Ma, H. Yang, M. R. Lyu, and I. King. Sorec: social recommendation using probabilistic matrix factorization. In *CIKM*, pages 931–940. ACM, 2008.

[83] H. Ma, D. Zhou, C. Liu, M. R. Lyu, and I. King. Recommender systems with social regularization. In *WSDM*, pages 287–296. ACM, 2011.

[84] H. Ma, T. C. Zhou, M. R. Lyu, and I. King. Improving recommender systems by incorporating social contextual information. *TOIS*, 29(2):9, 2011.

[85] P. Massa and P. Avesani. Trust-aware recommender systems. In *RecSys*, pages 17–24. ACM, 2007.

[86] P. Massa and P. Avesani. Trust-aware recommender systems. In *RecSys*, pages 17–24. ACM, 2007.

[87] C. E. Mello, M. A. Aufaure, and G. Zimbrao. Active learning driven by rating impact analysis. In *RecSys*, pages 341–344. ACM, 2010.

[88] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, pages 1928–1937, 2016.

[89] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv*, 2013.

[90] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[91] F. Monti, M. Bronstein, and X. Bresson. Geometric matrix completion with recurrent multi-graph neural networks. In *NIPS*, pages 3697–3707, 2017.

[92] M. Niepert, M. Ahmed, and K. Kutzkov. Learning convolutional neural networks for graphs. In *ICML*, pages 2014–2023, 2016.

[93] R. Pálovics, A. A. Benczúr, L. Kocsis, T. Kiss, and E. Frigó. Exploiting temporal influence in online recommendation. In *RecSys*, pages 273–280. ACM, 2014.

[94] R. Pan, Y. Zhou, B. Cao, N. N. Liu, R. Lukose, M. Scholz, and Q. Yang. One-class collaborative filtering. In *ICDM*, pages 502–511. IEEE, 2008.

[95] H. Pei, B. Wei, K. C. Chang, Y. Lei, and B. Yang. Geom-gcn: Geometric graph convolutional networks. In *ICLR*, 2020.

[96] S. Rendle and C. Freudenthaler. Improving pairwise learning for item recommendation from implicit feedback. In *WSDM*, pages 273–282. ACM, 2014.

[97] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. In *UAI*, pages 452–461, 2009.

[98] J. D. Rennie and N. Srebro. Fast maximum margin matrix factorization for collaborative prediction. In *ICML*, pages 713–719. ACM, 2005.

[99] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186. ACM, 1994.

[100] F. Ricci, L. Rokach, and B. Shapira. Introduction to recommender systems handbook. In *Recommender systems handbook*, pages 1–35. Springer, 2011.

[101] N. Rubens, M. Elahi, M. Sugiyama, and D. Kaplan. Active learning in recommender systems. In *Recommender systems handbook*, pages 809–846. Springer, 2015.

[102] R. Salakhutdinov and A. Mnih. Probabilistic matrix factorization. In *NIPS*, pages 1257–1264, 2007.

[103] R. Salakhutdinov and A. Mnih. Bayesian probabilistic matrix factorization using markov chain monte carlo. In *ICML*, pages 880–887. ACM, 2008.

[104] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *WWW*, pages 285–295. ACM, 2001.

[105] D. Sculley. Combined regression and ranking. In *SIGKDD*, pages 979–988. ACM, 2010.

[106] G. Shani, D. Heckerman, and R. I. Brafman. An mdp-based recommender system. *JMLR*, 6(Sep):1265–1295, 2005.

[107] Y. Shen and R. Jin. Learning personal+ social latent factor model for social recommendation. In *SIGKDD*, pages 1303–1311. ACM, 2012.

[108] J. Shi, Y. Yu, Q. Da, S. Chen, and A. Zeng. Virtual-taobao: Virtualizing real-world online retail environment for reinforcement learning. In *AAAI*, volume 33, pages 4902–4909, 2019.

[109] Y. Shi, M. Larson, and A. Hanjalic. List-wise learning to rank with matrix factorization for collaborative filtering. In *RecSys*, pages 269–272. ACM, 2010.

[110] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[111] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *ICML*, pages 387–395, 2014.

[112] D. Silver, L. Newnham, D. Barker, S. Weller, and J. McFall. Concurrent reinforcement learning from customer interactions. In *ICML*, pages 924–932, 2013.

[113] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[114] B. Song, X. Yang, Y. Cao, and C. Xu. Neural collaborative ranking. In *CIKM*, pages 1353–1362. ACM, 2018.

[115] W. Song, Z. Xiao, Y. Wang, L. Charlin, M. Zhang, and J. Tang. Session-based social recommendation via dynamic graph attention networks. In *WSDM*, pages 555–563. ACM, 2019.

[116] N. Srebro and T. Jaakkola. Weighted low-rank approximations. In *ICML*, pages 720–727, 2003.

[117] P. Sun, L. Wu, and M. Wang. Attentive recurrent social recommendation. In *SIGIR*, pages 185–194. ACM, 2018.

[118] Z. Sun, J. Yang, J. Zhang, A. Bozzon, L. Huang, and C. Xu. Recurrent knowledge graph embedding for effective recommendation. In *RecSys*, pages 297–305, 2018.

[119] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[120] N. Taghipour, A. Kardan, and S. S. Ghidary. Usage-based web recommendations: a reinforcement learning approach. In *RecSys*, pages 113–120. ACM, 2007.

[121] J. Tang, H. Gao, H. Liu, and A. Das Sarma. etrust: Understanding trust evolution in an online world. In *SIGKDD*, pages 253–261. ACM, 2012.

[122] J. Tang, X. Hu, and H. Liu. Social recommendation: a review. *Social Network Analysis and Mining*, 3(4):1113–1133, 2013.

[123] L. Tang, Y. Jiang, L. Li, C. Zeng, and T. Li. Personalized recommendation via parameter-free contextual bandits. In *SIGIR*, pages 323–332. ACM, 2015.

[124] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *AAAI*, 2016.

[125] H. Van Seijen, M. Fatemi, J. Romoff, R. Laroche, T. Barnes, and J. Tsang. Hybrid reward architecture for reinforcement learning. In *NIPS*, pages 5392–5402, 2017.

[126] H. P. Vanchinathan, I. Nikolic, F. De Bona, and A. Krause. Explore-exploit in top-n recommender systems via gaussian processes. In *RecSys*, pages 225–232. ACM, 2014.

[127] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.

[128] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv*, 2017.

[129] J. Vermorel and M. Mohri. Multi-armed bandit algorithms and empirical evaluation. In *ECML*, pages 437–448. Springer, 2005.

[130] P. Victor, C. Cornelis, M. De Cock, and P. P. Da Silva. Gradual trust and distrust in recommender systems. *Fuzzy Sets and Systems*, 160(10):1367–1382, 2009.

[131] M. Volkovs and R. S. Zemel. Collaborative ranking with 17 parameters. In *NIPS*, pages 2294–2302, 2012.

[132] H. Wang, X. Shi, and D. Yeung. Collaborative recurrent autoencoder: Recommend while learning to fill in the blanks. In *NIPS*, pages 415–423, 2016.

[133] H. Wang, N. Wang, and D. Yeung. Collaborative deep learning for recommender systems. In *SIGKDD*, pages 1235–1244. ACM, 2015.

[134] H. Wang, Q. Wu, and H. Wang. Learning hidden features for contextual bandits. In *CIKM*, pages 1633–1642. ACM, 2016.

[135] H. Wang, Q. Wu, and H. Wang. Factorization bandits for interactive recommendation. In *AAAI*, pages 2695–2702, 2017.

[136] H. Wang, F. Zhang, J. Wang, M. Zhao, W. Li, X. Xie, and M. Guo. Ripplenet: Propagating user preferences on the knowledge graph for recommender systems. In *CIKM*, pages 417–426, 2018.

[137] H. Wang, F. Zhang, M. Zhang, J. Leskovec, M. Zhao, W. Li, and Z. Wang. Knowledge-aware graph neural networks with label smoothness regularization for recommender systems. In *SIGKDD*, pages 968–977. ACM, 2019.

[138] H. Wang, F. Zhang, M. Zhao, W. Li, X. Xie, and M. Guo. Multi-task feature learning for knowledge graph enhanced recommendation. In *WWW*, pages 2000–2010, 2019.

[139] H. Wang, M. Zhao, X. Xie, W. Li, and M. Guo. Knowledge graph convolutional networks for recommender systems. In *WWW*, pages 3307–3313. ACM, 2019.

[140] J. Wang, A. P. De Vries, and M. J. Reinders. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *SIGIR*, pages 501–508. ACM, 2006.

[141] L. Wang, W. Zhang, X. He, and H. Zha. Supervised reinforcement learning with recurrent neural network for dynamic treatment recommendation. In *SIGKDD*, pages 2447–2456. ACM, 2018.

[142] X. Wang, X. He, M. Wang, F. Feng, and T. Chua. Neural graph collaborative filtering. In *SIGIR*. ACM, 2019.

[143] X. Wang, Y. Wang, D. Hsu, and Y. Wang. Exploration in interactive personalized music recommendation: a reinforcement learning approach. *TOMM*, 11(1):7, 2014.

[144] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[145] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[146] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[147] Q. Wu, H. Wang, Q. Gu, and H. Wang. Contextual bandits in a collaborative environment. In *SIGIR*, pages 529–538. ACM, 2016.

[148] Q. Wu, H. Wang, L. Hong, and Y. Shi. Returning is believing: Optimizing long-term user engagement in recommender systems. In *CIKM*, pages 1927–1936. ACM, 2017.

[149] Q. Wu, H. Zhang, X. Gao, P. He, P. Weng, H. Gao, and G. Chen. Dual graph attention networks for deep latent representation of multifaceted social effects in recommender systems. pages 2091–2102, 2019.

[150] S. Wu, Y. Tang, Y. Zhu, L. Wang, X. Xie, and T. Tan. Session-based recommendation with graph neural networks. In *AAAI*, pages 346–353, 2019.

[151] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[152] B. Yang, Y. Lei, J. Liu, and W. Li. Social collaborative filtering by trust. *TPAMI*, 39(8):1633–1647, 2017.

[153] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *SIGKDD*, pages 974–983. ACM, 2018.

[154] X. Yu, X. Ren, Y. Sun, Q. Gu, B. Sturt, U. Khandelwal, B. Norick, and J. Han. Personalized entity recommendation: A heterogeneous information network approach. In *WSDM*, pages 283–292, 2014.

[155] F. Zhang, N. J. Yuan, D. Lian, X. Xie, and W. Ma. Collaborative knowledge base embedding for recommender systems. In *SIGKDD*, pages 353–362, 2016.

[156] S. Zhang, L. Yao, A. Sun, and Y. Tay. Deep learning based recommender system: A survey and new perspectives. *CSUR*, 52(1):5, 2019.

[157] X. Zhao, L. Xia, L. Zhang, Z. Ding, D. Yin, and J. Tang. Deep reinforcement learning for page-wise recommendations. In *RecSys*, pages 95–103. ACM, 2018.

[158] X. Zhao, L. Zhang, Z. Ding, L. Xia, J. Tang, and D. Yin. Recommendations with negative feedback via pairwise deep reinforcement learning. In *SIGKDD*, pages 1040–1048. ACM, 2018.

[159] X. Zhao, W. Zhang, and J. Wang. Interactive collaborative filtering. In *CIKM*, pages 1411–1420. ACM, 2013.

[160] G. Zheng, F. Zhang, Z. Zheng, Y. Xiang, N. J. Yuan, X. Xie, and Z. Li. Drn: A deep reinforcement learning framework for news recommendation. In *WWW*, pages 167–176. IW3C2, 2018.

[161] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[162] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *International conference on algorithmic applications in management*, pages 337–348. Springer, 2008.

[163] L. Zou, L. Xia, P. Du, Z. Zhang, T. Bai, W. Liu, J. Nie, and D. Yin. Pseudo dyna-q: A reinforcement learning framework for interactive recommendation. In *WSDM*. ACM, 2020.