ADAPTIVE AND AUTOMATED DEEP RECOMMENDER SYSTEMS

By

Xiangyu Zhao

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science — Doctor of Philosophy

2021

ABSTRACT

ADAPTIVE AND AUTOMATED DEEP RECOMMENDER SYSTEMS

By

Xiangyu Zhao

Recommender systems are intelligent information retrieval applications, and have been leveraged in numerous domains such as e-commerce, movies, music, books, and point-of-interests. They play a crucial role in the users' information-seeking process, and overcome the information overload issue by recommending personalized items (products, services, or information) that best match users' needs and preferences. Driven by the recent advances in machine learning theories and the prevalence of deep learning techniques, there have been tremendous interests in developing deep learning based recommender systems. They have unprecedentedly advanced effectiveness of mining the non-linear user-item relationships and learning the feature representations from massive datasets, which produce great vitality and improvements in recommendations from both academic and industry communities.

Despite above prominence of existing deep recommender systems, their adaptiveness and automation still remain under-explored. Thus, in this dissertation, we study the problem of adaptive and automated deep recommender systems. Specifically, we present our efforts devoted to building adaptive deep recommender systems to continuously update recommendation strategies according to the dynamic nature of user preference, which maximizes the cumulative reward from users in the practical streaming recommendation scenarios. In addition, we propose a group of automated and systematic approaches that design deep recommender system frameworks effectively and efficiently from a data-driven manner. More importantly, we apply our proposed models to a variety of real-world recommendation platforms and have achieved promising enhancements of social and economic benefits.

To my parents for their love and support.

ACKNOWLEDGMENTS

This dissertation would have never been completed without the help, support, and guidance from many great people through my Ph.D. study at Michigan State University in the past four and a half years.

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Jiliang Tang for his best mentorship, encouragement, and support during my Ph.D. study. As an advisor, he has worked tirelessly to provide help, advice, guidance, and inspiration to me, which led me to grow into an independent scholar. Besides research, he is also my role model for life, and has taught me the value of kindness, optimism, ambition and responsibility through his words and actions. I could not have imagined having a better advisor and mentor for my Ph.D. study. I would also like to thank my committee members Prof. Pang-Ning Tan, Dr. Jiayu Zhou, Dr. Mi Zhang, and Dr. Dawei Yin, for their continuous advice and insightful comments.

In addition, I am thankful to all of my awesome lab-mates from the Data Science and Engineering (DSE) lab at Michigan State University: Tyler Derr, Zhiwei Wang, Yao Ma, Hamid Karimi, Wenqi Fan, Xiaorui Liu, Haochen Liu, Han Xu, Xiaoyang Wang, Jamell Dacon, Wentao Wang, Wei Jin, Yaxin Li, Yiqi Wang, Juanhui Li, Harry Shomer, Jie Ren, Jiayuan Ding, Haoyu Han, Hongzhi Wen, Yuxuan Wan, Hua Liu, and Norah Alfadhli. I enjoyed all stimulating discussions, exchange of ideas, and staying awake until the last moment of the paper deadlines. During my time in the DSE Lab I have learned so much from you all. I am also thankful for the collaboration and help from outside the DSE Lab: Prof. Jiquan Chen, Prof. Yi Chang, Dr. Charu Aggarwal, Dr. Taiquan Peng, Dr. Weinan Zhang, Dr. Ruiming Tang, Prof. Heng Huang, Dr. Hui Liu, Dr. Li Zhao, Dr. Grace Hui Yang, Dr. Alex

Beutel, Dr. Rui Chen, Dr. Jason Gauci, Dr. Minmin Chen, Dr. Shauki Jain, Dr. Yongfeng Zhang, Dr. Fei Sun, Prof. Jimmy Xiangji Huang, and Yingqiang Ge.

Moreover, I appreciate the experience of working with the extraordinary collaborators I met during internships. They are Dr. Dawei Yin, Dr. Long Xia, Dr. Lixin Zou, Dr. Liang Zhang, Dr. Zhaochun Ren, and Zhuoye Ding at JD.com; Dr. Xiwang Yang, Xiaobing Liu, Dr. Chong Wang, Changsheng Gu, Ming Chen, Xudong Zheng, Haoshenglun Zhang, Dr. Taiqing Wang, and Dr. Aonan Zhang at Bytedance; Dr. Bo Long, Dr. Bee-chung Chen, Dr. Huiji Gao, Dr. Weiwei Guo, Dr. Jun Shi and Sida Wang at LinkedIn. Without their precious support it would not be possible to conduct this research.

Lastly, I would again like to thank my parents, entire family and girlfriend for their love and support. This dissertation is dedicated to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

Introduction

1.1 Motivation and Challenges

Recommender systems are intelligent information retrieval applications. They assist users in their information-seeking tasks by suggesting items (products, services, or information) that best fit their needs and preferences. Recommender systems have become increasingly popular in recent years, and have been utilized in a variety of domains, including movies, music, books, search queries, and social tags [107, 108]. Typically, a recommendation procedure can be modeled as interactions between users and recommender agent (RA). It consists of two phases: 1) user model construction and 2) recommendation generation [90]. During the interaction, the recommender agent builds a user model to learn users' preferences based on users' personal information or historical behaviors. Then, the recommender agent generates a list of items that best match users' preferences.

Driven by the recent advances in deep learning, there have been increasing interests in developing deep learning based recommender systems (DLRSs) [154, 97, 142]. Architectures of DLRS often mainly consist of three key components: (i) *embedding layers* that map raw user/items features in a high dimensional space to dense vectors in a low dimensional *embedding space*, (ii) *hidden layers* that perform nonlinear transformations to transform the input features, and (iii) *output layers* that make predictions for specific recommendation

tasks (e.g. regression and classification) based on the representations from hidden layers. DLRSs have boosted the recommendation performance because of their capacity of effectively catching the non-linear user-item relationships, and learning the complex abstractions as data representations [154].

Most existing recommender systems are static and hand-crafted in (i) learning recommendation policy and (ii) designing DLRS framework. First, recommendation policy is the strategy that a recommender system suggests items to users, such as collaborative filtering methods recommend similar items to users of similar tastes [77]. To learn recommendation policy, there are two overarching issues:

- Most recommender systems consider the recommendation procedure as a static process and make recommendations following a fixed greedy strategy, which may fail given the dynamic nature of the users' preferences;
- Existing recommendation policies are designed to maximize the immediate reward of recommendations, i.e., to make users purchase the recommended items, while completely overlooking whether these items will lead to more profitable rewards in the long run.

Second, DLRS framework consists of all the key components of building a deep recommender system, such as data processing (e.g., data cleaning and feature engineering), model selection (e.g., W&D [24] v.s. DeepFM [51]), neural architecture (e.g., embedding, hidden and output layers), optimization (e.g., loss function, optimizer and learning rate) and model evaluation. To design DLRS framework, we face three inherent challenges:

- The majority of existing DLRSs are developed based on hand-crafted components, which requires ample expert knowledge of machine learning and recommender systems;

- Human error and bias can lead to suboptimal components, which reduces the recommendation effectiveness;
- Non-trivial time and engineering efforts are usually required to design the task-specific components in different recommendation scenarios.

In this dissertation, we propose to design adaptive and automated deep recommender systems to address challenges in the two aforementioned perspectives, i.e., recommendation policy and DLRS framework.

To learn adaptive recommendation policy, we will consider the recommendation procedure as sequential interactions between users and recommender agents; and leverage Reinforcement Learning (RL) to automatically learn an optimal recommendation strategy (policy) that maximizes cumulative rewards from users without any specific instructions. Recommender systems based on reinforcement learning have two advantages. First, they are able to continuously update their trial-and-error strategies during the interactions, until the system converges to the optimal strategy that generates recommendations best fitting their users' dynamic preferences. Second, the models in the system are trained via estimating the present value with delayed rewards under current states and actions. The optimal strategy is made by maximizing the expected long-term cumulative reward from users. Therefore, the system could identify items with small immediate rewards but making considerable contributions to the rewards for future recommendations.

To design automated DLRS framework, the recent development of automated machine learning (AutoML), easing the usage of machine learning tools and designing task-dependent learning models, has become an important and popular area with both practical needs and research values [81]. It has changed the convention of model design from manual to automatic,

provides unprecedented opportunities to design deep model components. Inspired by AutoML techniques, we will propose a set of unified and systematic approaches, which design the DLRS framework in an automated and data-driven manner [157, 158, 81, 100]. AutoML allows non-experts to build DLRS without requiring them to become experts in machine learning and recommender systems, excludes the possible human error and bias, and significantly reduces the computational and temporal cost.

1.2 Dissertation Contributions

The major contribution of this proposal are summarized as follows:

- We conduct pioneering research to develop adaptive and automated deep recommender systems from two perspectives: (i) recommendation policy and (ii) DLRS framework, which could dynamically and adaptively enhance the recommendation performance in the long run, and reduce the requirement of expert knowledge and human efforts to design the sophisticated DLRS frameworks;

- As users are typically recommended a page of items in real-world recommender systems, we propose a novel page-wise recommendation framework, which leverages deep reinforcement learning to automatically learn the optimal recommendation strategies and optimize a page of items simultaneously;

- Most platforms optimize recommending and advertising strategies by different teams separately via different techniques. We propose a two-level deep reinforcement learning framework with novel deep Q-network architectures to jointly optimize the user experience and advertising revenue in online recommender systems;

- Directly training and evaluating a new RL-based recommendation algorithm needs to collect users' feedback in real systems, which is time/effort consuming and could downgrade users' experiences. We propose to build a user simulator for RL-based recommendations, which can model real users' behaviors and can be used to pre-train and evaluate new recommendation algorithms before launching them online.

- Most recommender systems allocate a unified embedding dimension to all feature fields, which is memory inefficient. We propose a novel AutoML-based framework to automatically assign different embedding dimensions to different feature fields, which can achieve better recommendation performance with much fewer embedding parameters.

- Existing recommender systems often leverage a predefined and fixed loss function that could lead to suboptimal recommendation quality and training efficiency. We propose an AutoML-based framework to automatically select the appropriate loss function for different data examples according to their varied convergence behaviors, improving recommendation performance and training efficiency with excellent transferability.

1.3 Dissertation Overview

The remainder of this dissertation is organized as follows. In Chapter 2, we introduce an RL-based framework DeepPage for capturing users' dynamic preferences and learning the item display strategy within a page. Chapter 3 presents a two-level RL framework RAM, where the first level acts as the recommender system to select a subset of items from the large item space, and the second level serves as the advertising system to assign the right advertisement to the right user on the right place. In Chapter 4, we develop a user simulator UserSim based on a Generative Adversarial Network, where the generator captures the underlying

distribution of users' historical logs and generates realistic logs, while the discriminator not only distinguishes real and fake logs but also predicts users' behaviors. Chapter 5 presents our work on automatically allocating different embedding dimensions to different feature fields according to their contributions in recommendation. In Chapter 6, we introduce an AutoLoss framework to select proper loss function for each data example automatically, where we develop a novel controller network to dynamically adjust the loss probabilities in a differentiable manner. Finally, Chapter 7 concludes the dissertation and presents promising future research directions.

Chapter 2

Page-wise Recommendations

Abstract

Recommender systems can mitigate the information overload problem by suggesting users' personalized items. In real-world recommendations such as e-commerce, a typical interaction between the system and its users is – users are recommended a page of items and provide feedback; and then the system recommends a new page of items. To effectively capture such interaction for recommendations, we need to solve two key problems – (1) how to update recommending strategy according to user's *real-time feedback*, and (2) how to generate a page of items with proper display, which pose tremendous challenges to traditional recommender systems. In this chapter, we study the problem of page-wise recommendations aiming to address aforementioned two challenges simultaneously. In particular, we propose a principled approach to jointly generate a set of complementary items and the corresponding strategy to display them in a 2-D page; and propose a novel page-wise recommendation framework based on deep reinforcement learning, DeepPage, which can optimize a page of items with proper display based on real-time feedback from users. The experimental results based on a real-world e-commerce dataset demonstrate the effectiveness of the proposed framework.

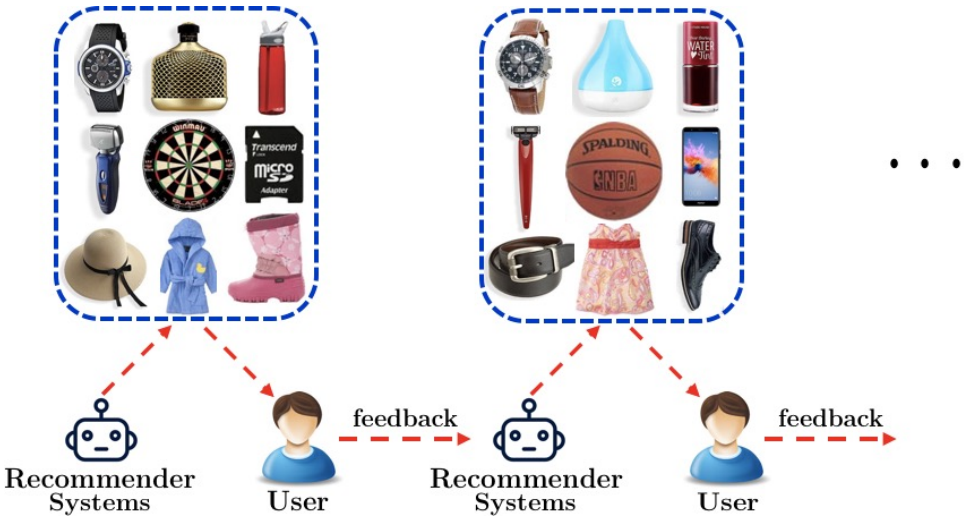

Figure 2.1: An example of interactions between recommender systems and users.

2.1 Introduction

Recommender systems play an essential role in the information-seeking process and overcome information overload by facilitating the interactions between business platforms and users. Figure 2.1 illustrates a typical example of the interactions between an e-commerce recommender system and a user – each time the system recommends a page of items to the user; next, the user browses these items and provides real-time feedback and then the system recommends a new page of items. This example suggests two key challenges to effectively take advantage of these interactions for e-commerce recommender systems – 1) how to efficiently capture user's dynamic preference and update recommending strategy according to user's real-time feedback; and 2) how to generate a page of items with proper display based on user's preferences.

2.1.1 Real-time Feedback

Most existing recommender systems consider the recommendation procedure as a static process and make recommendations following a fixed greedy strategy. However, these

approaches may fail to capture the dynamic nature of the users' preferences, and they become infeasible to efficiently and continuously update their recommending strategies according to user's real-time feedback. Thus, in this work, we consider the recommendation procedure as sequential interactions between users and the recommender agent; and leverage Reinforcement Learning (RL) to automatically learn the optimal recommendation strategies. Recommender systems based on reinforcement learning have two major advantages. First, they are able to continuously update their strategies based on user's real-time feedback during the interactions, until the system converges to the optimal strategy that generates recommendations best fitting users' dynamic preferences. Second, the optimal strategy is made by maximizing the expected long-term cumulative reward from users; while the majority of traditional recommender systems are designed to maximize the immediate (short-term) reward of recommendations [116]. Therefore, the system can identify items with small immediate rewards but making significant contributions to the rewards for future recommendations.

2.1.2 Page-wise Recommendations

As mentioned in the example, users are typically recommended a page of items. To achieve this goal, we introduce a page-wise recommender system, which is able to jointly (1) generate a set of diverse and complementary items and (2) form an item display strategy to arrange the items in a 2-D page that can lead to maximal reward. Conventional RL methods could recommend a set of items each time. For instance, DQN can recommend a set of items with the highest Q-values according to the current state[93]. However, these approaches recommend items based on the same state, which leads to the recommended items being similar. In practice, a bundling of complementary items may receive higher rewards than recommending all similar items. For instance, in real-time news feed recommendations, a user

may want to read diverse topics of interest[153]. In addition, page-wise recommendations need to properly display a set of generated items in a 2-D page. Traditional approaches treat it as a ranking problem, i.e., ranking items into a 1-D list according to the importance of items. In other words, user's most preferred item is posited at the top of list. However, in e-commerce recommender systems, a recommendation page is a 2-D grid rather than a 1-D list. Also, eye-tracking studies [121] show that rather than linearly scanning a page, users do page chunking, i.e., they partition the 2-D page into chunks, and browse the chunk they prefer more. In addition, the set of items and the display strategy are generated separately; hence they may not be optimal to each other. Therefore, page-wise recommendations need principled approaches to simultaneously generate a set of complementary items and the corresponding display strategy in a 2-D page.

2.1.3 Contributions

In this chapter, we tackle the two aforementioned challenges simultaneously by introducing a novel page-wise recommender system based on deep reinforcement learning. We summarize our major contributions as follows:

- We introduce a principled approach to generate a set of complementary items and properly display them in one 2-D recommendation page simultaneously;
- We propose a page-wise recommendation framework DeepPage, which can jointly optimize a page of items by incorporating real-time feedback from users;
- We demonstrate the effectiveness of the proposed framework in a real-world e-commerce dataset and validate the effectiveness of the components in DeepPage for accurate recommendations.

2.2 The Proposed Framework

In this section, we first give an overview of the proposed Actor-Critic based reinforcement learning recommendation framework with notations. Then we present the technical details of components in Actor and Critic, respectively.

2.2.1 Framework Overview

As mentioned in Section 2.1.1, we model the recommendation task as a Markov Decision Process (MDP) and leverage Reinforcement Learning (RL) to automatically learn the optimal recommendation strategies, which can continuously update recommendation strategies during the interactions and the optimal strategy is made by maximizing the expected long-term cumulative reward from users. In practice, conventional RL methods like POMDP[116] and Q-learning[125] become infeasible with the increasing number of items for recommendations. Thus, we leverage Deep Reinforcement Learning[75] with (adapted) artificial neural networks as the non-linear approximators to estimate the action-value function in RL. This model-free reinforcement learning method does not estimate the transition probability and store the Q-value table. Hence it can support a huge amount of items in recommender systems.

There are two major challenges when we apply deep reinforcement learning to the studied problem – (a) the large (or even continuous) and dynamic action space (item space), and (b) the computational cost to select an optimal action (a set of items). In practice, using only discrete indices to denote items is insufficient since we cannot know the relations between different items only from indices. One common way is to use extra information to represent items with continuous embeddings[69]. Besides, the action space of recommender systems is dynamic as items arriving and leaving. Moreover, computing the Q-value for all state-action

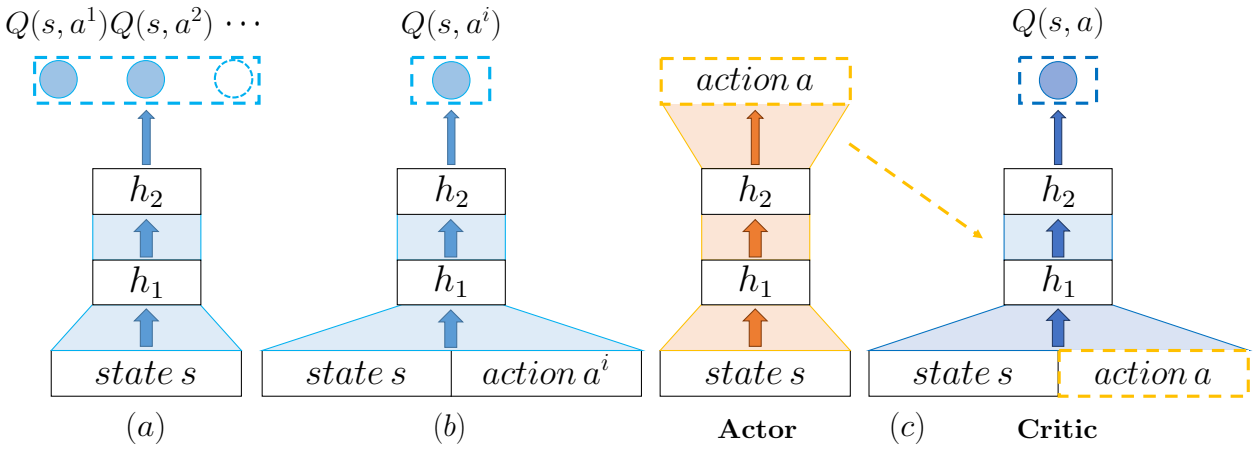

Figure 2.2: Framework architecture selection.

pairs is a time-consuming task because of the enormous state and action spaces.

To tackle these challenges, in this chapter, our recommending policy builds upon the Actor-Critic framework [124], shown in Figure 2.2 (c). The Actor-Critic architecture is preferred from the studied problem since it is suitable for large and dynamic action space, and can also reduce redundant computation simultaneously compared to alternative architectures as shown in Figures 2.2 (a) and (b). The conventional Deep Q-learning architectures shown in Figure 2.2 (a) inputs only the state space and outputs Q-values of all actions. This architecture is suitable for the scenario with high state space and small/fixed action space like Atari[93], but cannot handle large and dynamic action space scenarios, like recommender systems. Also, we cannot leverage the second conventional deep Q-learning architecture as shown in Figure 2.2(b) because of its temporal complexity. This architecture inputs a state-action pair and outputs the Q-value correspondingly, and makes use of the optimal action-value function $Q^*(s,a)$. It is the maximum expected return achievable by the optimal policy, and should follow the Bellman equation [6] as:

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a')|s,a\right] \tag{2.1}$$

In practice, selecting an optimal a', $|\mathcal{A}|$ evaluations is necessary for the inner operation "$\max_{a'}$". In other words, this architecture computes Q-value for all $a' \in \mathcal{A}$ separately, and then selects the maximal one. This prevents Eq. (2.1) from being adopted in practical recommender systems.

In the Actor-Critic framework, the Actor architecture inputs the current state s and aims to output a deterministic action (or recommending a deterministic page of M items), i.e., $s \rightarrow a = \{a^1, \cdots, a^M\}$. The Critic inputs only this state-action pair rather than all potential state-action pairs, which avoids the aforementioned computational cost as follows:

$$Q(s, a) = \mathbb{E}_{s'} \left[r + \gamma Q(s', a') | s, a \right] \tag{2.2}$$

where the Q-value function $Q(s, a)$ is a judgment of whether the selected action matches the current state, i.e., whether the recommendations match user's preference. Finally, according to the judgment from Critic, the Actor updates its' parameters in a direction of boosting recommendation performance so as to output proper actions in the next iteration. With the above design intuitions, we formally define the tuple of five elements $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ of MDP as follows:

- **State space** $\mathcal{S}$: A state $s \in \mathcal{S}$ is defined as user's current preference, which is generated based on user's browsing history, i.e., the items that a user browsed and her corresponding feedback.

- **Action space** $\mathcal{A}$: An action $a = \{a^1, \cdots, a^M\} \in \mathcal{A}$ is to recommend a page of M items to a user based on current state s.

- **Reward** $\mathcal{R}$: After the RA takes an action a at the state s, i.e., recommending a page

of items to a user, the user browses these items and provides her feedback. She can skip (not click), click, or order these items, and the agent receives immediate reward $r(s, a)$ according to the user's feedback.

- **Transition** $\mathcal{P}$: Transition $p(s'|s, a)$ defines the state transition from s to s' when RA takes action a.

- **Discount factor** γ: $\gamma \in [0, 1]$ defines the discount factor when we measure the present value of future reward. In particular, when $\gamma = 0$, RA only considers the immediate reward. In other words, when $\gamma = 1$, all future rewards can be counted fully into that of the current action.

Specifically, we model the recommendation task as an MDP in which a recommender agent (RA) interacts with environment $\mathcal{E}$ (or users) over a sequence of time steps. At each time step, the RA takes an action $a \in \mathcal{A}$ according to $\mathcal{E}$'s state $s \in \mathcal{S}$, and receives a reward $r(s, a)$ (or the RA recommends a page of items according to user's current preference, and receives user's feedback). As the consequence of action a, the environment $\mathcal{E}$ updates its state to s' with transition $p(s'|s, a)$. The goal of reinforcement learning is to find a recommendation policy $\pi : \mathcal{S} \to \mathcal{A}$, which can maximize the cumulative reward for the recommender system. Next, we will elaborate on the Actor and Critic architectures for the proposed framework.

2.2.2 Architecture of Actor Framework

The Actor is designed to generate a page of recommendations according to user's preference, which needs to tackle three challenges – 1) setting an initial preference at the beginning of a new recommendation session, 2) learning the real-time preference in the current session, which should capture the dynamic nature of user's preference in current session and spatial

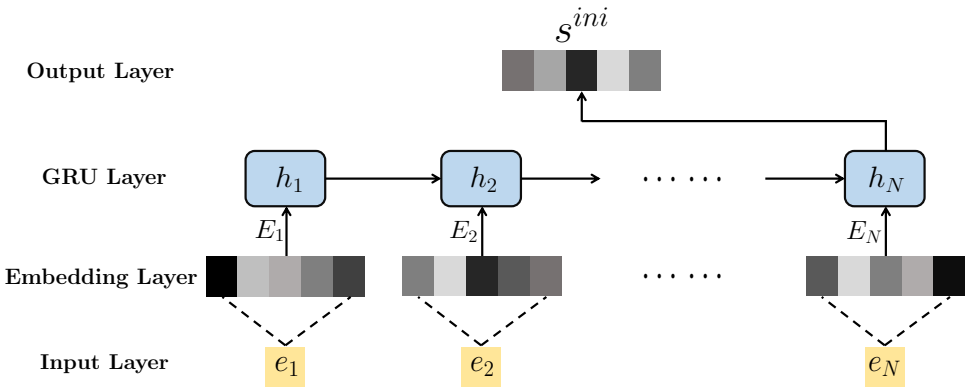

Figure 2.3: Encoder to generate initial state s^{ini}.

item display patterns in a page, and 3) jointly generating a set of recommendations and displaying them in a 2-D page. To address these challenges, we propose an Actor framework with the Encoder-Decoder architecture.

2.2.2.1 Encoder for Initial State Generation Process

Figure 2.3 illustrates the model for generating initial preference. We introduce a RNN with Gated Recurrent Units (GRU) to capture users' sequential behaviors as user's initial preference. The inputs of GRU are user's last clicked/ordered items $\{\boldsymbol{e}_1, \cdots, \boldsymbol{e}_N\}$ (sorted in chronological order) before the current session, while the output is the representation of users' initial preference by a vector. The input $\{\boldsymbol{e}_1, \cdots, \boldsymbol{e}_N\}$ is dense and low-dimensional vector representations of items [1]. We add an item-embedding layer to transform $\boldsymbol{e}_i$ into a low-dimensional dense vector via $\boldsymbol{E}_i = \tanh(\boldsymbol{W}_{\boldsymbol{E}}\boldsymbol{e}_i + \boldsymbol{b}_{\boldsymbol{E}}) \in \mathbb{R}^{|\boldsymbol{E}|}$ where we use "tanh" activate function since $\boldsymbol{e}_i \in (-1, +1)$.

We leverage GRU rather than Long Short-Term Memory (LSTM) because that GRU outperforms LSTM for capturing users' sequential preference in recommendation task [55].

[1] These item representations are pre-trained using users' browsing history by a company, i.e. each item is treated as a word and the clicked items in one recommendation session as a sentence, and item representations are trained via word embedding[69]. The effectiveness of these item representations is validated by their business such as searching, ranking, bidding and recommendations.

Unlike LSTM using input gate i_t and forget gate f_t to generate a new state, GRU utilizes an update gate z_t:

$$z_t = \sigma(\boldsymbol{W}_z \boldsymbol{E}_t + \boldsymbol{U}_z \boldsymbol{h}_{t-1}) \tag{2.3}$$

GRU leverages a reset gate r_t to control the input of the former state $\boldsymbol{h}_{t-1}$:

$$r_t = \sigma(\boldsymbol{W}_r \boldsymbol{E}_t + \boldsymbol{U}_r \boldsymbol{h}_{t-1}) \tag{2.4}$$

Then the activation of GRU is a linear interpolation between the previous activation $\boldsymbol{h}_{t-1}$ and the candidate activation $\hat{\boldsymbol{h}}_t$:

$$\boldsymbol{h}_t = (1 - z_t)\boldsymbol{h}_{t-1} + z_t \hat{\boldsymbol{h}}_t \tag{2.5}$$

where candidate activation function $\hat{\boldsymbol{h}}_t$ is computed as:

$$\hat{\boldsymbol{h}}_t = \tanh[\boldsymbol{W}\boldsymbol{E}_t + \boldsymbol{U}(r_t \cdot \boldsymbol{h}_{t-1})] \tag{2.6}$$

We use the final hidden state $\boldsymbol{h}_t$ as the representation of the user's initial state $\boldsymbol{s^{ini}}$ at the beginning of current recommendation session as:

$$\boldsymbol{s^{ini}} = \boldsymbol{h}_t. \tag{2.7}$$

2.2.2.2 Encoder for Real-time State Generation Process

Figure 2.4 illustrates the model to generate real-time preference in current session. In the page-wise recommender system, the inputs $\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_M\}$ for each recommendation page are

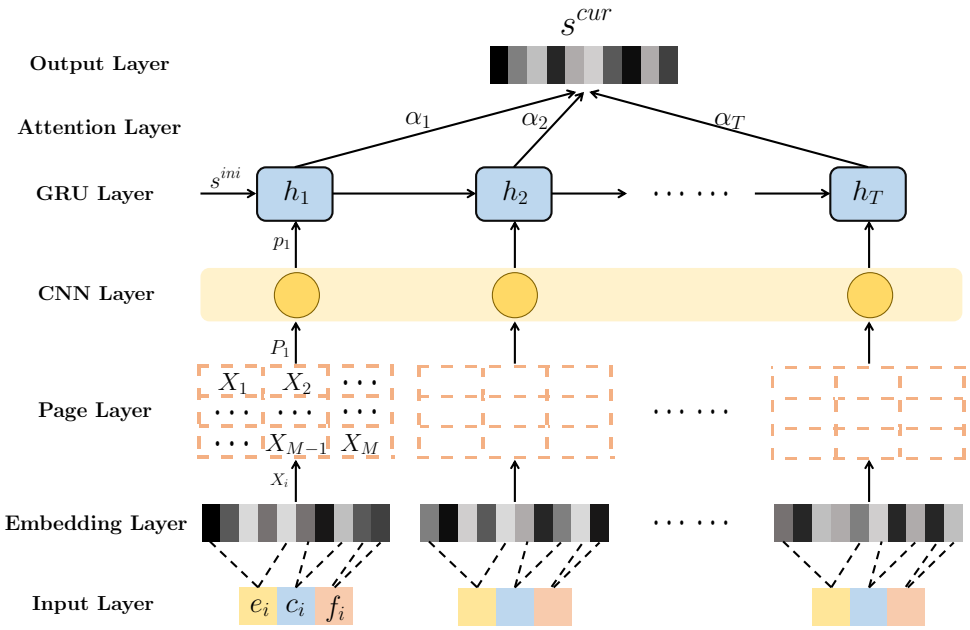

Figure 2.4: Encoder to generate current state s^{cur}.

the representations of the items in the page and user's corresponding feedback, where M is the size of a recommendation page and $\boldsymbol{x}_i$ is a tuple as:

$$\boldsymbol{x}_i = (\boldsymbol{e}_i, \boldsymbol{c}_i, \boldsymbol{f}_i), \tag{2.8}$$

where $\boldsymbol{e}_i$ is the aforementioned item representation. To assist the RA in capturing user's preference among different categories of items and generating complementary recommendations, we incorporate item's category $\boldsymbol{c}_i$. The item's category $\boldsymbol{c}_i$ is an one-hot indicator vector where $\boldsymbol{c}_i(i) = 1$ if this item belongs to the i^{th} category and other entities are zero. The one-hot indicator vector is extremely sparse and high-dimensional; hence we add a category-embedding layer transforming $\boldsymbol{c}_i$ into a low-dimensional dense vector $\boldsymbol{C}_i = \tanh(\boldsymbol{W}_{\boldsymbol{C}}\boldsymbol{c}_i + \boldsymbol{b}_{\boldsymbol{C}}) \in \mathbb{R}^{|\boldsymbol{C}|}$.

In addition to information from items, $\boldsymbol{e}_i$ and $\boldsymbol{c}_i$, we also want to capture user's interests or feedback in current recommendation page. Thus, we introduce user's feedback vector $\boldsymbol{f}_i$, which is a one-hot vector to indicate user's feedback for item i, i.e., skip/click/order. Similarly,

we transform $\boldsymbol{f}_i$ into a dense vector $\boldsymbol{F}_i = \tanh(\boldsymbol{W_F}\boldsymbol{f}_i + \boldsymbol{b_F}) \in \mathbb{R}^{|\boldsymbol{F}|}$ via the embedding layer. Finally, we get a low-dimensional dense vector $\boldsymbol{X}_i \in \mathbb{R}^{|\boldsymbol{X}|}$ ($|\boldsymbol{X}| = |\boldsymbol{E}| + |\boldsymbol{C}| + |\boldsymbol{F}|$) by concatenating $\boldsymbol{E}_i$, $\boldsymbol{C}_i$ and $\boldsymbol{F}_i$ as:

$$\begin{aligned} \boldsymbol{X}_i &= concat(\boldsymbol{E}_i, \boldsymbol{C}_i, \boldsymbol{F}_i) \\ &= concat\big(\tanh(\boldsymbol{W_E}\boldsymbol{e}_i + \boldsymbol{b_E}, \boldsymbol{W_C}\boldsymbol{c}_i + \boldsymbol{b_C}, \boldsymbol{W_F}\boldsymbol{f}_i + \boldsymbol{b_F})\big) \end{aligned} \tag{2.9}$$

Note that all item-embedding layers share the same parameters $\boldsymbol{W_E}$ and $\boldsymbol{b_E}$, which reduces the number of parameters and achieves better generalization. We apply the same constraints for category and feedback embeddings.

Then, we reshape the transformed item representations $\{\boldsymbol{X}_1, \cdots, \boldsymbol{X}_M\}$ as the original arrangement in the page. In other words, we arrange the item representations in one page $\boldsymbol{P}_t$ as 2D grids similar to one image. For instance, if one recommendation page has h rows and w columns ($M = h \times w$), we will get a $h \times w|\boldsymbol{X}|$ matrix $\boldsymbol{P}_t$. To learn item spatial display strategy in one page that leads to maximal reward, we introduce Convolutional Neural Network (CNN). CNN is a successful architecture in computer vision applications because of its capability to apply various learnable kernel filters on images to discover complex spatial correlations [67]. Hence, we utilize 2D-CNN followed by fully connected layers to learn the optimal item display strategy as:

$$\boldsymbol{p}_t = conv2d(\boldsymbol{P}_t), \tag{2.10}$$

where $\boldsymbol{p}_t$ is a low-dimensional dense vector representing the information from the items and user's feedback in page $\boldsymbol{P}_t$ as well as the spatial patterns of the item display strategy of page $\boldsymbol{P}_t$.

Next, we feed $\{\boldsymbol{p}_1, \cdots, \boldsymbol{p}_T\}$ into another RNN with Gated Recurrent Units (GRU) to capture user's real-time preference in the current session. The architecture of this GRU is similar to the one in Section 2.2.2.1, but we utilize the final hidden state $\boldsymbol{s}^{ini}$ in Section 2.2.2.1 as the initial state in current GRU. Furthermore, to capture the user's real-time preference in the current session, we employ attention mechanism [3], which allows the RA to adaptively focus on and linearly combine different parts of the input sequence:

$$\boldsymbol{s}^{cur} = \sum_{t=1}^{T} \alpha_t \boldsymbol{h}_t \tag{2.11}$$

where the weighted factors α_t determine which parts of the input sequence should be emphasized or ignored when making predictions. Here we leverage location-based attention mechanism [89] where the weighted factors are computed from the target hidden state $\boldsymbol{h}_t$ as follows:

$$\alpha_t = \frac{exp(\boldsymbol{W}_{\boldsymbol{\alpha}} \boldsymbol{h}_t + \boldsymbol{b}_{\boldsymbol{\alpha}})}{\sum_j exp(\boldsymbol{W}_{\boldsymbol{\alpha}} \boldsymbol{h}_j + \boldsymbol{b}_{\boldsymbol{\alpha}})} \tag{2.12}$$

This GRU with attention mechanism is able to dynamically select more important input, which is helpful to capture the user's real-time preference in the current session. Note that - 1) the length of this GRU is flexible according to that of the current recommendation session. After each user-agent interaction, i.e., the user browses one page of generated recommendations and give feedback to RA, we can add one more GRU unit, and use this page of items, corresponding categories and feedback as the input of the new GRU unit; 2) in fact, the two RNNs in Section 2.2.2.1 and Section 2.2.2.2 can be integrated into one RNN, we describe them separately to clearly illustrate their architecture, and validate their effectiveness in Section 2.4.4.

2.2.2.3 Decoder for Action Generation Process

In this subsection, we will propose the action $\boldsymbol{a}^{cur}$ generation process, which generates (recommends) a new page of items to users. In other words, given user's current preference $\boldsymbol{s}^{cur}$, we aim to recommend a page of items and display them properly to maximize the reward. It is the inverse process of what the convolutional layer does. Hence, we use Deconvolution Neural Network (DeCNN) [147] to restore one page from the low-dimensional representation $\boldsymbol{s}^{cur}$. It provides a sophisticated and automatic way to generate a page of recommendations with the corresponding display as:

$$\boldsymbol{a}^{cur} = deconv2d(\boldsymbol{s}^{cur}). \tag{2.13}$$

Note that - 1) the size of $\boldsymbol{a}^{cur}$ and $\boldsymbol{P}$ are different, since $\boldsymbol{a}^{cur}$ only contains item-embedding $\boldsymbol{E}_i$, while $\boldsymbol{P}$ also contains item's category embedding $\boldsymbol{C}_i$ and feedback-embedding $\boldsymbol{F}_i$. For instance, if one recommendation page has h rows and w columns ($M = h \times w$), $\boldsymbol{P}$ is a $h \times w|\boldsymbol{X}|$ matrix, while $\boldsymbol{a}^{cur}$ is a $h \times w|\boldsymbol{E}|$ matrix; and 2) the generated item embeddings in $\boldsymbol{a}^{cur}$ may be not in the real item embedding set, thus we need to map them to valid item embeddings, which will be provided in later sections.

2.2.3 The Architecture of Critic Framework

The Critic is designed to leverage an approximator to learn an action-value function $Q(s, a)$, which is a judgment of whether the action (or a recommendation page) a generated by Actor matches the current state s. Note that we use "s" as $\boldsymbol{s}^{cur}$ in the last subsection for simplicity. Then, according to $Q(s, a)$, the Actor updates its' parameters in the direction of improving performance to generate proper actions (or recommendations) in the following iterations.

Thus we need to feed user's current state s and action a (or a recommendation page) into the critic. To generate user's current state s, The RA follows the same strategy from Eq. (2.3) to Eq. (2.11), which uses embedding layers, 2D-CNN and GRU with attention mechanism to capture user's current preference. For action a, because $\boldsymbol{a}^{cur}$ generated in Eq. (2.13) is a 2D matrix similar to an image, we utilize the same strategy in Eq. (2.10), a 2D-CNN, to degrade $\boldsymbol{a}^{cur}$ into a low-dimensional dense vector a as:

$$a = conv2d(\boldsymbol{a}^{cur}). \tag{2.14}$$

Then the RA concatenates current state s and action a, and feeds them into a Q-value function $Q(s, a)$. In real recommender systems, the state and action spaces are enormous, thus estimating the action-value function $Q(s, a)$ for each state-action pair is infeasible. In addition, many state-action pairs may not appear in the real trace such that it is hard to update their values. Therefore, it is more flexible and practical to use an approximator function to estimate the action-value function. In practice, the action-value function is usually highly non-linear. Thus we choose Deep neural networks as approximators. In this work, we refer to a neural network approximator as a deep Q-value function (DQN).

2.3 Training and Test Procedure

In this section, we discuss the training and test procedures. We propose two policies, i.e., online-policy and off-policy, to train and test the proposed framework based on online environment and offline historical data, respectively. Off-policy is necessary because the proposed framework should be pre-trained offline and be evaluated before launching them

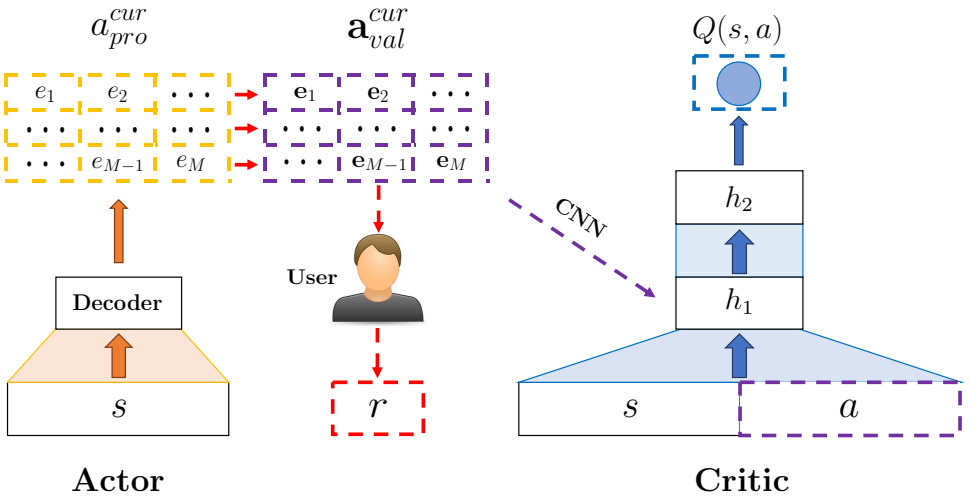

Figure 2.5: An illustration of the proposed framework.

online to ensure the quality of the recommendations and mitigate possible negative impacts on user experience. After the offline stage, we can apply the framework online, and then the framework can continuously improve its strategies during the interactions with users.

2.3.1 The Training Procedure

As aforementioned in Section 2.2.2, we map user's preference $\boldsymbol{s}^{cur}$ to a new page of recommendations ($\boldsymbol{a}^{cur}$). In a page of M items, $\boldsymbol{a}^{cur}$ contains item-embeddings of M items, i.e., $\{\boldsymbol{e}_1, \cdots, \boldsymbol{e}_M\}$. However, $\boldsymbol{a}^{cur}$ is a *proto-action*, because the generated item embedding $\boldsymbol{e}_i \in \boldsymbol{a}^{cur}$ may be not in the existing item-embedding space $\mathcal{I}$. Therefore, we need to map from proto-action $\boldsymbol{a}_{pro}^{cur}$ to *valid-action* $\boldsymbol{a}_{val}^{cur}$ where we have $\{\boldsymbol{e}_i \in \mathcal{I} | \forall \{\boldsymbol{e}_i \in \boldsymbol{a}_{val}^{cur}\}$. With this modification, an illustration of the proposed Actor-Critic recommending framework is demonstrated in Figure 2.5 where we omit Encoders for state generation part.

2.3.1.1 Online Training Procedure

When we train the proposed framework in online environment, RA can interact with users by sequentially choosing recommendation items over a sequence of time steps. Thus, in online environment, the RA is able to receive real-time feedback for the recommended items from users. In this setting, for each $\boldsymbol{e}_i$ in $\boldsymbol{a}_{pro}^{cur}$, we select the most similar $\boldsymbol{e}_i \in \mathcal{I}$ as the *valid item-embedding* in $\boldsymbol{a}_{val}^{cur}$. In this work, we select *cosine similarity* as:

$$\begin{aligned} \boldsymbol{e}_i &= \arg\max_{\boldsymbol{e}\in\mathcal{I}} \frac{\boldsymbol{e}_i^\top \cdot \boldsymbol{e}}{\|\boldsymbol{e}_i\|\|\boldsymbol{e}\|} \\ &= \arg\max_{\boldsymbol{e}\in\mathcal{I}} \boldsymbol{e}_i^\top \cdot \frac{\boldsymbol{e}}{\|\boldsymbol{e}\|} \end{aligned} \tag{2.15}$$

To decrease the computational cost of Eq. (2.15), we pre-compute $\frac{\boldsymbol{e}}{\|\boldsymbol{e}\|}$ for all $\boldsymbol{e} \in \mathcal{I}$ and adopt item recalling mechanism to reduce the number of relevant items [2] .

Algorithm 2.1: Mapping Algorithm.

Input: User's browsing history s_h, item-embedding space $\mathcal{I}$, the size of recommendation page M.
Output: Valid recommendation page $\boldsymbol{a}_{val}^{cur}$.

1: Generate proto-action $\boldsymbol{a}_{pro}^{cur}$ according Eq. (2.3) to Eq. (2.11)
2: **for** $m = 1, M$ **do**
3: Select the most similar item as $\boldsymbol{e_m}$ according to Eq. (2.15)
4: Add item $\boldsymbol{e}_m$ into $\boldsymbol{a}_{val}^{cur}$ (at the same location as $\boldsymbol{e}_m$ in $\boldsymbol{a}_{pro}^{cur}$)
5: Remove item $\boldsymbol{e}_m$ from $\mathcal{I}$
6: **end for**
7: **return** $\boldsymbol{a}_{val}^{cur}$

We present the mapping algorithm in Algorithm 2.1. The Actor first generates proto-action $\boldsymbol{a}_{pro}^{cur}$ (line 1). For each $\boldsymbol{e}_m$ in $\boldsymbol{a}_{pro}^{cur}$, the RA selects the most similar item in terms of

[2] In general, user's preference in current session should be related to user's last clicked/ordered items before the current session(say $\mathcal{L}$). Thus for each item in $\mathcal{L}$, we collect a number of most similar items in terms of cosine similarity from the whole item space, and combine all collected items as the initial item-embedding space $\mathcal{I}$ of current recommendation session. During the current session, when a user clicks or orders an item, we will also add a number of most similar items into the item-embedding space $\mathcal{I}$.

cosine similarity (line 3), and then adds this item into $\boldsymbol{a}_{val}^{cur}$ at the same position as $\boldsymbol{e}_m$ in $\boldsymbol{a}_{pro}^{cur}$ (line 4). Finally, the RA removes this item from the item-embedding space (line 5), which prevents recommending the same item repeatedly in one recommendation page.

Then the RA recommends the new recommendation page $\boldsymbol{a}_{val}^{cur}$ to user, and receives the immediate feedback (reward) from user. The reward r is the summation of rewards of all items in this page:

$$r = \sum_{m=1}^{M} reward(\boldsymbol{e}_m) \tag{2.16}$$

2.3.1.2 Offline Training Procedure

When we use user's historical browsing data to train the proposed Actor-Critic framework, user's browsing history, the new recommendation page $\boldsymbol{a}_{val}^{cur}$ and user's corresponding feedback (reward) r are given in the data. Thus, there is a gap between $\boldsymbol{a}_{pro}^{cur}$ and $\boldsymbol{a}_{val}^{cur}$, i.e., no matter what proto-action $\boldsymbol{a}_{pro}^{cur}$ outputted by the Actor, the valid-action $\boldsymbol{a}_{val}^{cur}$ is fixed. This will disconnect the Actor and the Critic.

From existing work [75, 32] and Section 2.3.1.1, we learn that $\boldsymbol{a}_{pro}^{cur}$ and $\boldsymbol{a}_{val}^{cur}$ should be similar, which is the prerequisite to connect the Actor and the Critic for training. Thus, we choose to minimize the difference between $\boldsymbol{a}_{pro}^{cur}$ and $\boldsymbol{a}_{val}^{cur}$:

$$\min_{\theta^{\pi}} \sum_{b=1}^{B} \left(\|\boldsymbol{a}_{pro}^{cur} - \boldsymbol{a}_{val}^{cur}\|_F^2 \right) \tag{2.17}$$

where B is the batch size of samples in each iteration of SGD. Eq. (2.17) updates Actor's parameters in the direction of pushing $\boldsymbol{a}_{pro}^{cur}$ and $\boldsymbol{a}_{val}^{cur}$ to be similar. In each iteration, given user's browsing history, the new recommendation page $\boldsymbol{a}_{val}^{cur}$, the RA generates proto-action $\boldsymbol{a}_{pro}^{cur}$ and then minimizes the difference between $\boldsymbol{a}_{pro}^{cur}$ and $\boldsymbol{a}_{val}^{cur}$, which can connect the Actor

and the Critic. Next, we can follow conventional methods to update the parameters of Actor and Critic. The reward r is the summation of rewards of all items in page $\boldsymbol{a}_{val}^{cur}$.

2.3.1.3 Training Algorithm

In this work, we utilize DDPG [75] algorithm to train the parameters of the proposed Actor-Critic framework. The Critic can be trained by minimizing a sequence of loss functions $L(\theta^{\mu})$ as:

$$L(\theta^{\mu}) = \mathbb{E}_{s,a,r,s'}\big[\big(r + \gamma Q_{\theta^{\mu'}}(s', f_{\theta^{\pi'}}(s')) - Q_{\theta^{\mu}}(s,a)\big)^2\big] \tag{2.18}$$

where θ^{μ} represents all parameters in Critic. The critic is trained from samples stored in a replay buffer [94]. Actions stored in the replay buffer are generated by valid-action $\boldsymbol{a}_{val}^{cur}$, i.e., $a = conv2d(\boldsymbol{a}_{val}^{cur})$. This allows the learning algorithm to leverage the information of which action was actually executed to train the critic [32].

The first term $y = r + \gamma Q_{\theta^{\mu'}}(s', f_{\theta^{\pi'}}(s'))$ in Eq. (2.18) is the target for the current iteration. The parameters from the previous iteration $\theta^{\mu'}$ are fixed when optimizing the loss function $L(\theta^{\mu})$. In practice, it is often computationally efficient to optimize the loss function by stochastic gradient descent, rather than computing the full expectations in the above gradient. The derivatives of loss function $L(\theta^{\mu})$ with respective to parameters θ^{μ} are presented as follows:

$$\nabla_{\theta^{\mu}} L(\theta^{\mu}) = \mathbb{E}_{s,a,r,s'}\big[(r + \gamma Q_{\theta^{\mu'}}(s', f_{\theta^{\pi'}}(s')) - Q_{\theta^{\mu}}(s,a)) \nabla_{\theta^{\mu}} Q_{\theta^{\mu}}(s,a)\big] \tag{2.19}$$

We update the Actor using the policy gradient:

$$\nabla_{\theta^{\pi}} f_{\theta^{\pi}} \approx \mathbb{E}_s\big[\nabla_{\hat{a}} Q_{\theta^{\mu}}(s,\hat{a})\, \nabla_{\theta^{\pi}} f_{\theta^{\pi}}(s)\big] \tag{2.20}$$

where $\hat{a} = f_{\theta\pi}(s)$, i.e., $\hat{a}$ is generated by proto-action $\boldsymbol{a}_{pro}^{cur}$ ($\hat{a} = conv2d(\boldsymbol{a}_{pro}^{cur})$). Note that proto-action $\boldsymbol{a}_{pro}^{cur}$ is the actual action outputted by Actor. This guarantees that policy gradient is taken at the actual output of policy $f_{\theta\pi}$ [32].

The online training algorithm for the proposed framework DeepPage is presented in Algorithm 2.2. In each iteration, there are two stages, i.e., 1) transition generating stage (lines 7-10), and 2) parameter updating stage (lines 11-16). For transition generating stage (line 7): given the current state s_t, the RA first recommends a page of items a_t according to Algorithm 2.1 (line 8); then the RA observes the reward r_t and updates the state to s_{t+1} (lines 9); and finally the RA stores transitions (s_t, a_t, r_t, s_{t+1}) into the replay buffer $\mathcal{D}$ (line 10). For parameter updating stage (line 11): the RA samples mini-batch of transitions (s, a, r, s') from $\mathcal{D}$ (line 12), and then updates parameters of Actor and Critic (lines 13-16) following a standard DDPG procedure [75].

The offline training procedure is similar with Algorithm 2.2. The two differences are: 1) in line 8, offline training follows off-policy $b(s_t)$, and 2) before line 13, offline training first minimizes the difference between $\boldsymbol{a}_{pro}^{cur}$ and $\boldsymbol{a}_{val}^{cur}$ according to Eq. (2.17).

In the algorithm, we introduce widely used techniques to train our framework. For example, we utilize a technique known as *experience replay* [76] (lines 12), and introduce separated evaluation and target networks [93] (lines 2,16), which can help smooth the learning and avoid the divergence of parameters. For the soft updates of target networks (lines 16), we used $\tau = 0.01$. Moreover, we leverage *prioritized sampling strategy* [96] to assist the framework learning from the most important historical transitions.

Algorithm 2.2: Parameters Online Training for DeepPage with DDPG.

Initialize actor network f_{θ^π} and critic network Q_{θ^μ} with random weights
Initialize target network $f_{\theta^{\pi'}}$ and $Q_{\theta^{\mu'}}$ with weights $\theta^{\pi'} \leftarrow \theta^\pi, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize the capacity of replay buffer $\mathcal{D}$
for $session = 1, G$ **do**
  Receive initial observation state s_1
  for $t = 1, T$ **do**
    Stage 1: Transition Generating Stage
    Select an action a_t according to Alg. 2.1 (policy f_{θ^π})
    Execute action a_t and observe the reward r_t according to Eq. (2.16) and new state s_{t+1} according to Section 2.2.2.2
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Stage 2: Parameter Updating Stage
    Sample minibatch of $\mathcal{N}$ transitions (s, a, r, s') from $\mathcal{D}$
    Set $y = r + \gamma Q_{\theta^{\mu'}}(s', f_{\theta^{\pi'}}(s'))$
    Update Critic by minimizing $\frac{1}{\mathcal{N}} \sum_n \left(y - Q_{\theta\mu}(s, a)\right)^2$ according to Eq. (2.19)
    Update Actor using the sampled policy gradient according to Eq. (2.20)
    Update the target networks:

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

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

  end for
end for

Algorithm 2.3: Online Test for DeepPage.

1: Initialize Actor with the trained parameters Θ^{π}
2: Receive initial observation state s_1
3: **for** $t = 1, T$ **do**
4: Execute an action a_t according to Alg. 2.1 (policy $f_{\Theta^{\pi}}$)
5: Observe the reward r_t from user according to Eq. (2.16)
6: Observe new state s_{t+1} according to Section 2.2.2.2
7: **end for**

2.3.2 The Test Procedure

After the training stage, the proposed framework learns parameters Θ^{π} and Θ^{μ}. Now we formally present the test procedure of the proposed framework DeepPage. We design two test methods, i.e., 1) Online test: to test DeepPage in online environment where RA interacts with users and receives real-time feedback for the recommended items from users, and 2) Offline test: to test DeepPage based on user's historical browsing data.

2.3.2.1 Online Test

The online test algorithm in one recommendation session is presented in Algorithm 2.3. The online test procedure is similar to the transition generating stage in Algorithm 2.2. In each iteration of the recommendation session, given the current state s_t, the RA recommends a page of recommendations a_t to user following policy $f_{\Theta^{\pi}}$ (line 4). Then the RA observes the reward r_t from user (line 5) and updates the state to s_{t+1} (line 6).

2.3.2.2 Offline Test

The intuition of our offline test method is that, for a given recommendation session (offline data), the RA reranks the items in this session. If the proposed framework works well, this session's clicked/ordered items will be ranked at the top of the new list. The reason why RA

Algorithm 2.4: Offline Test of DeepPage Framework.

Input: Item set $\mathcal{I} = \{\boldsymbol{e}_1, \cdots, \boldsymbol{e}_N\}$ and corresponding reward set $\mathcal{R} = \{r_1, \cdots, r_N\}$ of a session.

Output:Recommendation list $\mathcal{L}$ with new order

1: Initialize Actor with well-trained parameters Θ^π
2: Receive initial observation state s_1
3: **while** $|\mathcal{I}| > 0$ **do**
4: Select an action a_t according to Alg. 2.1 (policy f_{Θ^π})
5: **for** $\boldsymbol{e}_i \in a_t$ **do**
6: Add $\boldsymbol{e}_i$ into the end of $\mathcal{L}$
7: Record reward r_i from user's historical browsing data
8: **end for**
9: Compute the overall reward r_t of a_t according to Eq. (2.16)
10: Execute action a_t and observe new state s_{t+1} according to Section 2.2.2.2
11: Remove all $\boldsymbol{e}_i \in a_t$ from $\mathcal{I}$
12: **end while**

only reranks items in this session rather than items in the whole item space is that for the offline dataset, we only have the ground truth rewards of the existing items in this session. The offline test algorithm in one recommendation session is presented in Algorithm 2.4. In each iteration of an offline test recommendation session, given the state s_t (line 2), the RA recommends a page of recommendations a_t following policy f_{Θ^π} (lines 4). For each item $\boldsymbol{e}_i$ in a_t, we add it into new recommendation list $\mathcal{L}$ (line 6), and record $\boldsymbol{e}_i$'s reward r_i from user's historical browsing data (line 7). Then we can compute the overall reward r_t of a_t (line 9) and update the state to s_{t+1} (line 10). Finally, we remove all items $\boldsymbol{e}_i$ in a_t from the item set $\mathcal{I}$ of the current session (line 11).

2.4 Experiments

In this section, we conduct extensive experiments with a dataset from a real e-commerce company to evaluate the effectiveness of the proposed framework. We mainly focus on two questions: (1) how the proposed framework performs compared to representative baselines;

and (2) how the components in Actor and Critic contribute to the performance. We first introduce experimental settings. Then we seek answers to the above two questions. Finally, we study the impact of important parameters on the performance of the proposed framework.

2.4.1 Experimental Settings

We evaluate our method on a dataset of September, 2017 from a real e-commerce company. We randomly collect 1,000,000 recommendation sessions (9,136,976 items) in temporal order, and use the first 70% sessions as the training/validation set and the later 30% sessions as the test set. For a new session, the initial state is collected from the previous sessions of the user. In this work, we leverage $N = 10$ previously clicked/ordered items to generate the initial state. Each time the RA recommends a page of $M = 10$ items (5 rows and 2 columns) to users [3]. The reward r of one skipped/clicked/ordered item is empirically set as 0, 1, and 5, respectively. The dimensions of item-embedding/ category-embedding/ feedback-embedding are $|\boldsymbol{E}| = 50$, $|\boldsymbol{C}| = 35$, and $|\boldsymbol{F}| = 15$. We set the discounted factor $\gamma = 0.95$, and the rate for soft updates of target networks $\tau = 0.01$. For the parameters of the proposed framework such as γ, we select them via cross-validation. Correspondingly, we also do parameter-tuning for baselines for a fair comparison. We will discuss more details about parameter selection for the proposed framework in the following subsections.

For online test, we leverage the summation of all rewards in one recommendation session as the metric. For offline test, we select **Precision@20**, **Recall@20**, **F1-score@20** [48], **NDCG@20** [59] and **MAP** [131], as the metrics to measure the performance. Our difference from traditional Learn-to-Rank methods is that we rank both clicked and ordered items

[3] This is based on offline historical data collected from mobile Apps, i.e., to fit the screen size of mobile phones, one page has only 5 rows and 2 columns.

together and set them by different rewards, rather than only rank clicked items as that in the Learn-to-Rank setting.

2.4.2 Performance Comparison for Offline Test

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

- **CF** [10]: Collaborative filtering is a method of making automatic predictions about the interests of a user by collecting preference information from many users, which is based on the hypothesis that people often get the best recommendations from someone with similar tastes to themselves.

- **FM** [106]: Factorization Machines combine the advantages of support vector machines with factorization models. Compared with matrix factorization, higher order interactions can be modeled using the dimensionality parameter.

- **GRU** [55] : This baseline utilizes the Gated Recurrent Units (GRU) to predict what user will click/order next based on the browsing histories. To make a fair comparison, it also keeps previous $N = 10$ clicked/ordered items as initial states.

- **DQN** [93]: We use a Deep Q-network with five fully-connected layers in this baseline. The input is the concatenation of embeddings of users' historical clicked/ordered items (state) and a page of recommendations (action), and train this baseline by Eq. (2.1).

- **DDPG** [32]: In this baseline, we use conventional Deep Deterministic Policy Gradient with five fully connected layers in both Actor and Critic. The input for Actor is the concatenation of embeddings of users' historical clicked/ordered items (state). The input

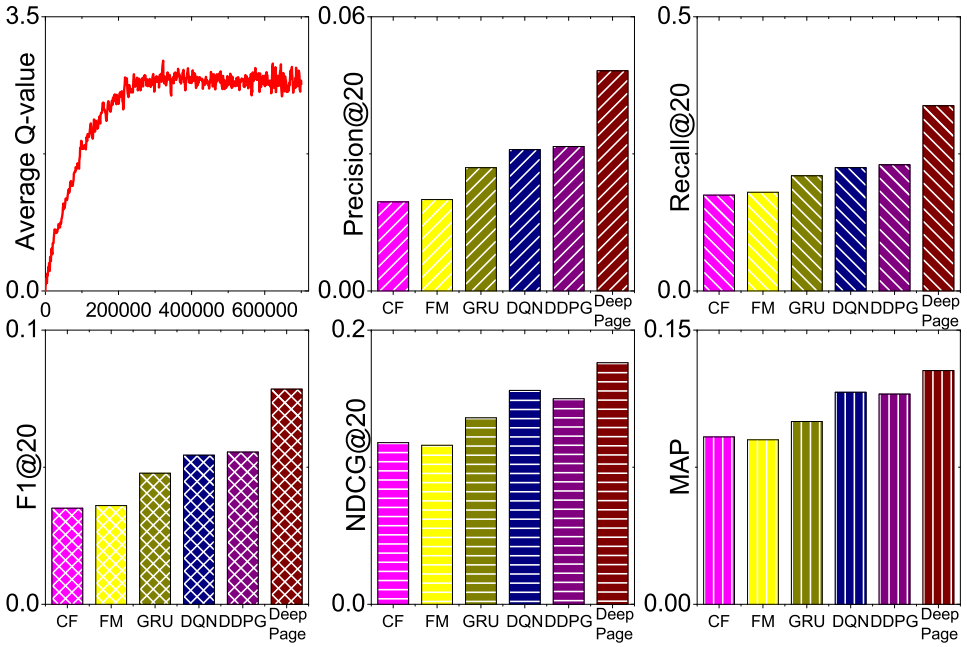

Figure 2.6: Overall performance comparison in offline test.

for Critic is the concatenation of embeddings of state and a page of recommendations (action).

We leverage offline training strategy to train DDPG and DeepPage as mentioned in Section 2.3.1.1. The results are shown in Figure 2.6. We make following observations:

- Figure 2.6 (a) illustrates the training process of DeepPage. We can observe that the framework approaches convergence when the model is trained by 500,000 offline sessions.
- CF and FM perform worse than other baselines. These two baselines ignore the temporal sequence of the users' browsing history, while GRU can capture the temporal sequence, and DQN, DDPG and DeepPage are able to continuously update their strategies during the interactions.
- DQN and DDPG outperform GRU. We design GRU to maximize the immediate reward for recommendations, while DQN and DDPG are designed to achieve the trade-off between

short-term and long-term rewards. This result suggests that introducing reinforcement learning can improve the performance of recommendations.

- DeepPage performs better than conventional DDPG. Compared to DDPG, DeepPage jointly optimizes a page of items and uses GRU to learn user's real-time preferences. More details about the impact of mode components on DeepPage will be discussed in the following subsection.

2.4.3 Performance Comparison for Online Test

Following [32], we build a simulated online environment (adapted to our case) for online test. We compare DeepPage with GRU, DQN and DDPG. Note that we do not include **CF** and **FM** baselines since CF and FM are not applicable to the online environment. To answer the second question, we systematically eliminate the corresponding components of the simulator by defining following variants of RecSimu:

Here we utilize online training strategy to train DDPG and DeepPage (both Actor-Critic framework) as mentioned in Section 2.3.1.2. Baselines are also applicable to be trained via the rewards generated by simulated online environment. Note that we use data different from the training set to build the simulated online environment.

As the test stage is based on the simulator, we can artificially control the length of recommendation sessions to study the performance in short and long sessions. We define short sessions with 10 recommendation pages, while long sessions with 50 recommendation pages. The results are shown in Figure 2.7. It can be observed:

- DDPG performs similar to DQN, but the training speed of DDPG is much faster than DQN, as shown in Figure 2.7 (a). DQN computes Q-value for all potential actions, while

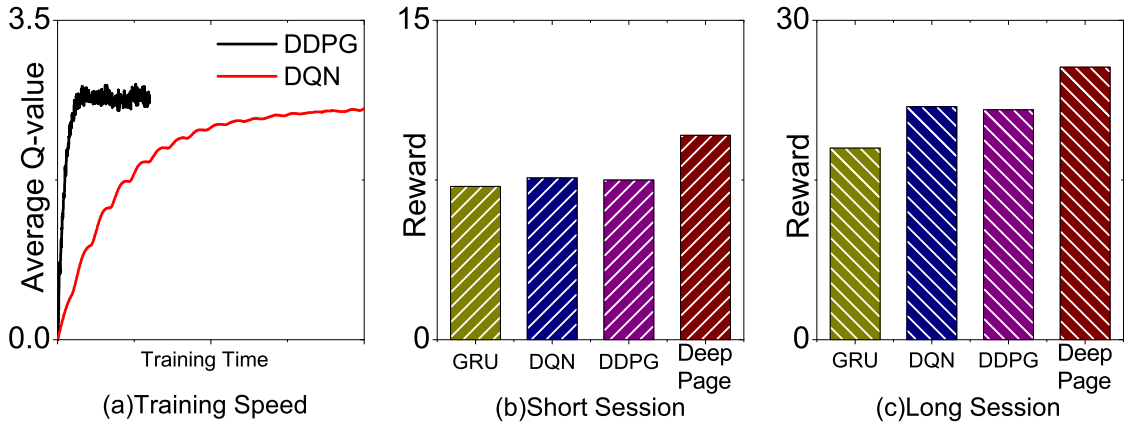

Figure 2.7: Overall performance comparison in online test.

DDPG can reduce this redundant computation. This result indicates that Actor-Critic framework is suitable for practical recommender systems with enormous action space.

- In short recommendation sessions, GRU, DQN and DDPG achieve comparable performance. In other words, GRU model and reinforcement learning models like DQN and DDPG can recommend proper items matching users' short-term interests.

- In long recommendation sessions, DQN and DDPG outperform GRU significantly. GRU is designed to maximize the immediate reward for recommendations, while reinforcement learning models like DQN and DDPG are designed to achieve the trade-off between short-term and long-term rewards. This result suggests that introducing reinforcement learning can improve the performance of recommendations.

- DeepPage performs better than conventional DQN and DDPG. DeepPage can learn user's real-time preferences and optimizes a page of items. We detail the effect of model components of DeepPage in the following subsection.

2.4.4 Effectiveness of Components

To validate the effectiveness of each component, we systematically eliminate the corresponding model components by defining following variants of DeepPage:

- DeepPage-1: This variant is to evaluate the performance of the embedding layer. We remove the embedding layer for items, categories and feedback.
- DeepPage-2: In this variant, we evaluate the contribution of category and feedback information, hence, this variant does not contain one-hot indicator vectors of category and feedback.
- DeepPage-3: This variant is to evaluate the effectiveness of GRU to generate initial state, so we eliminate the GRU in Figure 2.3.
- DeepPage-4: In this variant, we evaluate the contribution of CNNs as shown in Figure 2.4, thus we remove CNNs and directly feed the outputs of embedding layers (concatenate embeddings of all items as one vector) into GRU units.
- DeepPage-5: This variant is to evaluate the effectiveness of attention mechanism in Figure 2.4. Therefore, we eliminate attention layer and use the hidden state of last GRU unit as the input of DeCNN.
- DeepPage-6: In this variant, we evaluate the GRU to generate local state; thereby, we remove this GRU in Figure 2.4 and concatenate outputs of all CNNs as a vector, and feed it into DeCNN.
- DeepPage-7: This variant is to evaluate the performance of DeCNN to generate a new page of items; hence, we replace it with fully-connected layers, which output a concatenated

Table 2.1: Performance comparison of different components.

Methods	Precision @20	Recall @20	F1score @20	NDCG @20	MAP
DeepPage-1	0.0479	0.3351	0.0779	0.1753	0.1276
DeepPage-2	0.0475	0.3308	0.0772	0.1737	0.1265
DeepPage-3	0.0351	0.2627	0.0578	0.1393	0.1071
DeepPage-4	0.0452	0.3136	0.0729	0.1679	0.1216
DeepPage-5	0.0476	0.3342	0.0775	0.1716	0.1243
DeepPage-6	0.0318	0.2433	0.0528	0.1316	0.1039
DeepPage-7	0.0459	0.3179	0.0736	0.1698	0.1233
DeepPage	**0.0491**	**0.3576**	**0.0805**	**0.1872**	**0.1378**

vector of M item-embeddings.

The offline results are shown in Table 2.1 (we omit similar online observations because of the space limitation). We make following observations:

- DeepPage-1 and DeepPage-2 validate that incorporating category/feedback information and the embedding layer can boost recommendation performance.
- DeepPage-3 and DeepPage-6 perform worse, which suggests that setting user's initial preference at the beginning of a new recommendation session, and capturing user's real-time preference in current session is helpful for accurate recommendations.
- DeepPage-5 proves that incorporating attention mechanism can better capture user's real-time preference than only GRU.
- DeepPage outperforms DeepPage-4 and DeepPage-7, which indicates that item display strategy can influence the decision-making process of users.

In a nutshell, DeepPage outperforms all its variants, demonstrating each component's effectiveness for recommendations.

2.5 Related Work

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

The first category related to this chapter is traditional recommendation techniques. Recommender systems assist users by supplying a page of items that might interest users. Efforts have been made to offer meaningful recommendations to users. Collaborative filtering[77] is the most successful and the most widely used technique, which is based on the hypothesis that people often get the best recommendations from someone with similar tastes to themselves[10]. Another common approach is content-based filtering [95], which tries to recommend items with similar properties to those that a user ordered in the past. Knowledge-based systems[1] recommend items based on specific domain knowledge about how certain item features meet users' needs and preferences and how the item is useful for the user. Hybrid recommender systems are based on the combination of the above-mentioned two or more types of techniques[12]. The other topic closely related to this category is deep learning based recommender system, which is able to effectively capture the non-linear and non-trivial user-item relationships, and enables the codification of more complex abstractions as data representations in the higher layers[154]. For instance, Nguyen et al.[97] proposed a personalized tag recommender system based on CNN. It utilizes a constitutional and max-pooling layer to get visual features from patches of images. Wu et al.[142] designed a session-based recommendation model for real-world e-commerce website. It utilizes the basic RNN to predict what users will buy next based on the click histories.

The second category is about reinforcement learning for recommendations and personalization, which is different from the traditional item recommendations. In this work, we

consider the recommending procedure as sequential interactions between users and the recommender agent; and leverage deep reinforcement learning to automatically learn the optimal recommendation strategies. Indeed, reinforcement learning has been widely examined in the recommendation field. The MDP-Based CF model in Shani et al.[116] can be viewed as approximating a partial observable MDP (POMDP) by using a finite rather than unbounded window of past history to define the current state. Furthermore, Mahmood et al.[91] adopted the reinforcement learning technique to observe the responses of users in a conversational recommender, intending to maximize a numerical cumulative reward function modeling the benefit that users get from each recommendation session. Taghipour et al.[126, 125] modeled web page recommendation as a Q-Learning problem and learned to make recommendations from web usage data as the actions rather than discovering explicit patterns from the data. The system inherits the intrinsic characteristic of reinforcement learning, which is in a constant learning process. Sunehag et al.[122] introduced agents that successfully address sequential decision problems with high-dimensional combinatorial slate-action spaces. Zheng et al.[167] proposed a reinforcement learning framework to make online personalized news recommendation, which can effectively model the dynamic news features and user preferences, and plan for future explicitly, in order to achieve higher reward (e.g., CTR) in the long run. Feng et al.[38] presented a multi-agent reinforcement learning model, MA-RDPG, which can optimize ranking strategies collaboratively for multi-scenario ranking problems. Cai et al.[14] employed a reinforcement mechanism design framework for solving the impression allocation problem of large e-commerce websites, while taking the rationality of sellers into account.

Chapter 3

Jointly Recommend and Advertise

Abstract

Online recommendation and advertising are two major income channels for online recommendation platforms (e.g., e-commerce and news feed site). However, most platforms optimize recommending and advertising strategies by different teams separately via different techniques, leading to suboptimal overall performances. To this end, in this chapter, we propose a novel two-level reinforcement learning framework to jointly optimize the recommending and advertising strategies, where the first level generates a list of recommendations to optimize user experience in the long run; then the second level inserts ads into the recommendation list that can balance the immediate advertising revenue from advertisers and the negative influence of ads on long-term user experience. To be specific, the first level tackles high combinatorial action space problem that selects a subset of items from the large item space; while the second level determines three internally related tasks, i.e., (i) whether to insert an ad, and if yes, (ii) the optimal ad and (iii) the optimal location to insert. The experimental results based on real-world data demonstrate the effectiveness of the proposed framework.

3.1 Introduction

Practical e-commerce or news-feed platforms generally expose a hybrid list of recommended and advertised items (e.g., products, services, or information) to users, where recommending and advertising algorithms are typically optimized by different metrics [38]. The recommender systems (RS) capture users' implicit preferences from historical behaviors (e.g., clicks, rating and review) and generate a set of items that best match users' preferences. Thus, RS aims at optimizing the user experience or engagement. While advertising systems (AS) assign the right ad to the right user on the right ad slots to maximize the revenue, click-through rate (CTR) or return on investment (ROI) from advertisers. Thus, optimizing recommending and advertising algorithms independently may lead to suboptimal overall performance since exposing more ads to increase advertising revenue negatively influences user experience, and vice versa. Therefore, there is an increasing demand for developing a uniform framework that jointly optimizes recommending and advertising, so as to optimize the overall performance [156].

Efforts have been made to display recommended and advertised items together. They consider ads as recommendations, and rank all items in a hybrid list to optimize the overall ranking score [134]. However, this approach has two major drawbacks. First, solely maximizing the overall ranking score may result in suboptimal advertising revenue. Second, in the real-time bidding (RTB) environment, the vickrey-clarke-groves (VCG) mechanism is necessary to calculate the bid of each ad in the list, which suffers from many practical severe problems [112]. Therefore, it calls for methods where we can optimize not only the metrics for RS and AS separately, but also the overall performance. Moreover, more practical mechanisms such as generalized-second-price (GSP) should be considered to compute the bid of each ad.

To achieve the goal, we propose to study a two-level framework for rec-ads mixed display.

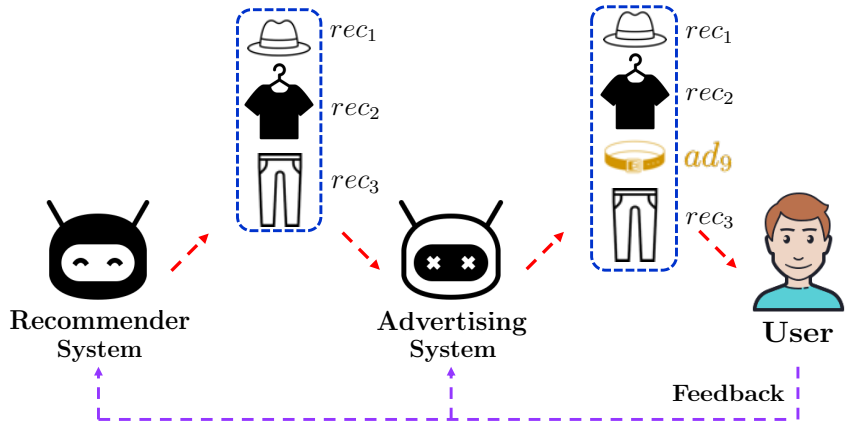

Figure 3.1: An example of rec-ads mixed display for one user request.

Figure 3.1 illustrates the high-level idea about how the framework works. Upon a user's request, the first level (i.e., RS) generates a list of recommendations (rec-list) according to the user's historical behaviors, aiming to optimize the long-term user experience or engagement. The main challenge to build the first level is the high computational complexity of the combinatorial action space, i.e., selecting a subset of items from the large item space. Then the second level (i.e., AS) inserts ads into the rec-list generated from the first level, and it needs to make three decisions, i.e., whether to insert an ad into the given rec-list; and if yes, the AS also needs to decide which ad and where to insert. For example, in Figure 3.1, the AS decides to insert an belt ad ad_9 between T-shirt rec_2 and pants rec_3 of the rec-list. The optimal ad should jointly maximize the immediate advertising revenue from advertisers in the RTB environment and minimize the negative influence of ads on user experience in the long run. Finally, the target user browses the mixed rec-ads list and provides her/his feedback. According to the feedback, the RS and AS update their policies and generate the mixed rec-ads list for the next iteration.

Most existing supervised learning based recommending and advertising methods are designed to maximize the immediate (short-term) reward and suggest items following fixed

greedy strategies. They overlook the long-term user experience and revenue. Thus, we build a two-level reinforcement learning framework for **R**ec/**A**ds **M**ixed display (RAM), which can continuously update their recommending and advertising strategies during the interactions with users, and the optimal strategy is designed to maximize the expected long-term cumulative reward from users [159, 155]. Meanwhile, to effectively leverage users' historical behaviors from other policies, we design an off-policy training approach, which can pre-train the framework before launching it online, so as to reduce the bad user experience in the initial online stage when new algorithms have not been well learned [161, 162, 175]. We summarize our major contributions as follows:

- We provide a two-stage generation process for the mixed display of recommended and advertised items in a hybrid list;
- We propose a two-level deep reinforcement learning based framework RAM, where the first level can generate a list of recommendations, while the second level can determine whether to insert an ad, and the corresponding optimal ad and location to insert;
- We conduct experiments with real-world data to demonstrate the effectiveness of the proposed framework.

3.2 Problem Statement

As aforementioned in Section 3.1, we consider the rec/ads mixed display task as a two-level reinforcement learning problem, and model it as a Markov Decision Process (MDP) where the RS and AS sequentially interact with users (environment $\mathcal{E}$) by generating a sequence of rec-ads hybrid-list over time, so as to maximize the cumulative reward from $\mathcal{E}$. Next, we

define the five elements $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ of the MDP.

- **State space $\mathcal{S}$**: A state $s_t \in \mathcal{S}$ includes a user's recommendation and advertisement browsing history before time t and the contextual information of current request at time t. The generated rec-list from RS is also considered as a part of the state for the AS.

- **Action space $\mathcal{A}$**: $a_t = (a_t^{rs}, a_t^{as}) \in \mathcal{A}$ is the action of RS and AS, where a_t^{rs} of RS is to generate a rec-list, and a_t^{as} of AS is to determine three internally related decisions, i.e., whether to insert an ad in current rec-list; and if yes, the AS needs to choose a specific ad and insert it into the optimal location of the rec-list. We denote $\mathcal{A}_t^{rs}$ and $\mathcal{A}_t^{as}$ as the rec and ad candidate sets for time t, respectively. Without the loss of generality, we assume that the length of any rec-list is fixed and the AS can insert at most one ad into a rec-list.

- **Reward $\mathcal{R}$**: After an action a_t is executed at the state s_t, a user browses the mixed rec-ads list and provides her feedback. The RS and AS will receive the immediate reward $r_t(s_t, a_t^{rs})$ and $r_t(s_t, a_t^{as})$ based on user's feedback. We will discuss more details about the reward in following sections.

- **Transition probability $\mathcal{P}$**: $P(s_{t+1}|s_t, a_t)$ is the state transition probability from s_t to s_{t+1} after executing action a_t. The MDP is assumed to satisfy $P(s_{t+1}|s_t, a_t, ..., s_1, a_1) = P(s_{t+1}|s_t, a_t)$.

- **Discount factor γ**: Discount factor $\gamma \in [0, 1]$ balances between current and future rewards – (1) $\gamma = 1$: all future rewards are fully considered into current action; and (2) $\gamma = 0$: only the immediate reward is counted.

Figure 3.2 illustrates the user-agent interactions in MDP. With the above definitions, the problem can be formally defined as follows: *Given the historical MDP, i.e.,* $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$,

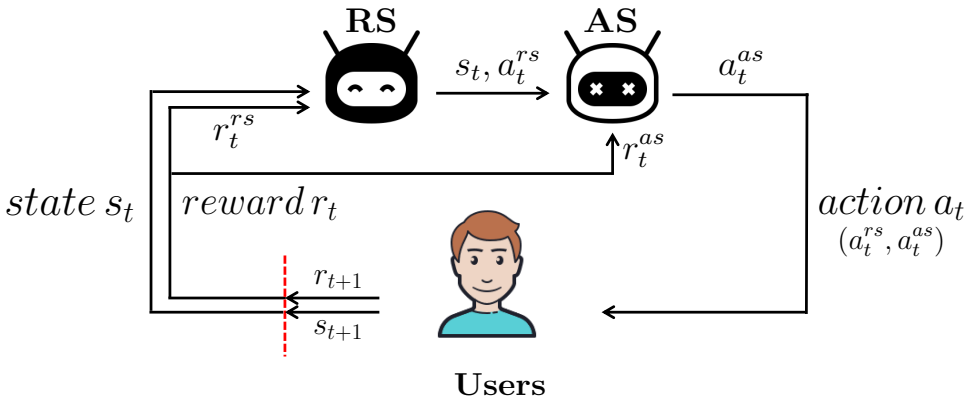

Figure 3.2: The agent-user interactions in MDP.

the goal is to find a two-level rec/ads policy $\pi = \{\pi_{rs}, \pi_{as}\} : \mathcal{S} \rightarrow \mathcal{A}$*, which can maximize the cumulative reward from users*, i.e., simultaneously optimizing the user experience and the advertising revenue.

3.3 Framework

In this section, we will discuss the two-level deep reinforcement learning framework for rec/ads mixed display. We will first introduce the first-level deep Q-network (i.e., RS) to generate a list of recommendations (rec-list) according to user's historical behaviors, then we propose a novel DQN architecture as the second-level (i.e., AS) to insert ads into the rec-list generated from RS. Finally, we discuss how to train the framework via offline data.

3.3.1 Deep Q-network for Recommendations

Given a user request, RS will return a list of items according to user's historical behaviors, which have two major challenges: (i) the high computational complexity of the combinatorial action space $\binom{|\mathcal{A}_t^{rs}|}{k}$, i.e., selecting k items from the large item space $\mathcal{A}_t^{rs}$, and (ii) how to approximate the action-value function (Q-function) for a list of items in the two-level

reinforcement learning framework. In this subsection, we introduce and enhance a cascading version of Deep Q-network to tackle the above challenges. Next, we first introduce the processing of state and action features, and then we illustrate the cascading Deep Q-network with an optimization algorithm.

3.3.1.1 The Processing of State and Action Features for RS

As mentioned in Section 3.2, a state s_t includes a user's rec/ads browsing history, and the contextual information of the current request. The browsing history contains a sequence of recommended items and a sequence of advertised items the user has browsed. Two RNNs with GRU (gated recurrent unit) are utilized to capture users' preferences of recommendations and advertisements, separately. The final hidden state of RNN is used to represent user's preference of recommended items p_t^{rec} (or ads p_t^{ad}). The contextual information c_t of current user request includes app version, operation system (e.g., ios and android) and feed type (swiping up/down the screen), etc. The state s_t is the concatenation p_t^{rec}, p_t^{ad} and c_t as:

$$s_t = concat(p_t^{rec}, p_t^{ad}, c_t) \tag{3.1}$$

For the transition from s_t to s_{t+1}, the browsed recommended and advertised items at time t will be inserted into the bottom of p_t^{rs} and p_t^{as} and we have p_{t+1}^{rs} and p_{t+1}^{as}, respectively. For the action $a_t^{rs} = \{a_t^{rs}(1), \cdots, a_t^{rs}(k)\}$ is the embedding of the list of k items that will be displayed in current request. Next, we will detail the cascading Deep Q-network.

3.3.1.2 The Cascading DQN for RS

Recommending a list of k items from the large item space $\mathcal{A}_t^{rs}$ is challenging because (i) the combinatorial action space $\binom{|\mathcal{A}_t^{rs}|}{k}$ has high computational complexity, and (ii) the order of items in the list also matters [160]. For example, a user may have different feedback to the same item if it is placed in different positions of the list. To resolve the above challenges, we leverage a cascading version of DQN which generates a list by sequentially selecting items in a cascading manner [23]. Given state s_t, the optimal action is denoted as:

$$a_t^{rs*} = \{a_t^{rs*}(1), \cdots, a_t^{rs*}(k)\} = \arg\max_{a_t^{rs}} Q^*(s_t, a_t^{rs}) \tag{3.2}$$

The key idea of the cascading DQN is inspired by the fact that:

$$\max_{a_t^{rs}(1:k)} Q^*\left(s_t, a_t^{rs}(1{:}k)\right) = \max_{a_t^{rs}(1)} \left(\max_{a_t^{rs}(2:k)} Q^*\left(s_t, a_t^{rs}(1{:}k)\right) \right) \tag{3.3}$$

which implies a cascade of mutually consistent as:

$$\begin{aligned}
a_t^{rs*}(1) &= \arg\max_{a_t^{rs}(1)} \left\{ Q^{1*}\left(s_t, a_t^{rs}(1)\right) := \max_{a_t^{rs}(2:k)} Q^*\left(s_t, a_t^{rs}(1{:}k)\right) \right\} \\
a_t^{rs*}(2) &= \arg\max_{a_t^{rs}(2)} \left\{ Q^{2*}\left(s_t, a_t^{rs*}(1), a_t^{rs}(2)\right) := \max_{a_t^{rs}(3:k)} Q^*\left(s_t, a_t^{rs}(1{:}k)\right) \right\} \\
&\cdots \\
a_t^{rs*}(k) &= \arg\max_{a_t^{rs}(k)} \left\{ Q^{k*}\left(s_t, a_t^{rs*}(1{:}k{-}1), a_t^{rs}(k)\right) := Q^*\left(s_t, a_t^{rs}(1{:}k)\right) \right\}
\end{aligned} \tag{3.4}$$

By applying above functions in a cascading fashion, we can reduce the computational complexity of obtaining the optimal action from $O\binom{|\mathcal{A}_t^{rs}|}{k}$ to $O(k|\mathcal{A}_t^{rs}|)$. Then the RS can sequentially select items following above equations. Note that the items already recommended

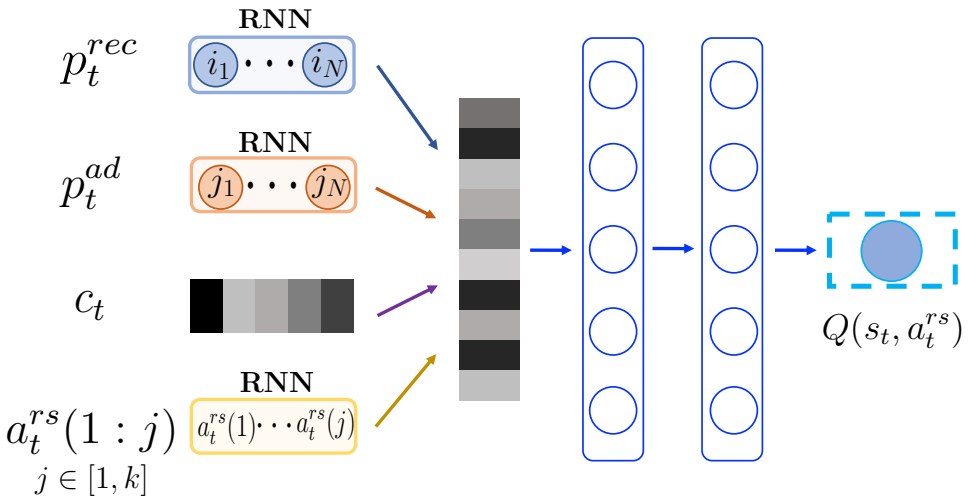

Figure 3.3: The architecture of cascading DQN for RS.

in the recommendation session will be removed from being recommended again. Next, we will detail how to estimate $\{Q^{j*}|j \in [1,k]\}$.

3.3.1.3 The estimation of Cascading Q-functions

Figure 3.3 illustrates the architecture of the cascading DQN, where $i_1, \cdots, i_N$ and $j_1, \cdots, j_N$ are uses' rec and ads browsing histories. The original model in [23] uses k layers to process k items separately without efficient weights sharing, which is crucial in handling large action size [118]. To address this challenge, we replace the k separate layers by RNN with GRU, where the input of j^{th} RNN unit is the feature of the j^{th} item in the list, and the final hidden state of RNN is considered as the representation of the list. Since all RNN units share the same parameters, the framework is flexible to any action size k.

To ensure that the cascading DQN selects the optimal action, i.e., a sequence of k optimal sub-actions, $\{Q^{j*}|j \in [1,k]\}$ functions should satisfy a set of constraints as follows:

$$Q^{j*}\left(s_t, a_t^{rs*}(1:j)\right) = Q^*\left(s_t, a_t^{rs*}(1:k)\right), \quad \forall j \in [1,k] \tag{3.5}$$

i.e., the optimal value of Q^{j*} should be equivalent to Q^* for all j. The cascading DQN enforces the above constraints in a soft and approximate way, where the loss functions are defined as follows:

$$\begin{aligned} &\left(y_t^{rs} - Q^j\left(s_t, a_t^{rs}(1:j)\right)\right)^2, \text{ where} \\ &y_t^{rs} = r_t\left(s_t, a_t^{rs}(1:k)\right) + \gamma Q^*\left(s_{t+1}, a_{t+1}^{rs*}(1:k)\right), \forall j \in [1,k] \end{aligned} \tag{3.6}$$

i.e., all Q^j functions fit against the same target y_t^{rs}. Then we update the parameters of the cascading DQN by performing gradient steps over the above loss. We will detail the reward function $r_t\left(s_t, a_t^{rs}(1:k)\right)$ in the following subsections. Next, we will introduce the second-level DQN for advertising.

3.3.2 Deep Q-network for Online Advertising

As mentioned in Section 3.1, the advertising system (AS) is challenging. First, AS needs to make three decisions, i.e., whether, which and where to insert. Second, these three decisions are dependent and traditional DQN architectures cannot be directly applied. For example, only when the AS decides to insert an ad, AS also needs to determine the candidate ads and locations. Third, the AS needs to not only maximize the income of ads but also minimize the negative influence on user experience. To tackle these challenges, next we detail a novel Deep Q-network architecture.

3.3.2.1 The Processing of State and Action Features for AS

We leverage the same architecture as that in Section 3.3.1.1 to obtain the state s_t. Furthermore, since the task of AS is to insert ad into a given rec-list, the output of the first-level DQN, i.e.,

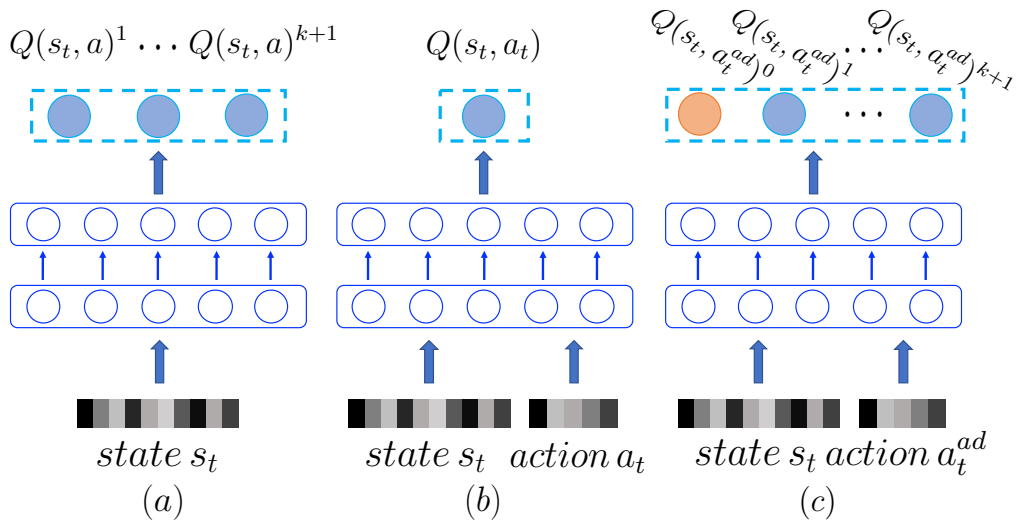

Figure 3.4: (a)(b) Two conventional DQNs. (c) Overview of the proposed DQN.

the current rec-list $a_t^{rs} = \{a_t^{rs}(1), \cdots, a_t^{rs}(k)\}$, is also considered as a part of the state for AS. For the action $a_t^{as} = (a_t^{ad}, a_t^{loc})$ of AS, a_t^{ad} is the embedding of a candidate ad. Given the rec-list of k items, there exist $k+1$ possible locations. Thus, we use a one hot vector $a_t^{loc} \in \mathbb{R}^{k+1}$ to indicate the location to insert the selected ad.

3.3.2.2 The Proposed DQN Architecture

Given state s_t and rec-list a_t^{rs}, the action of AS a_t^{as} contains three sub-actions, i.e., (i) whether to insert an ad into rec-list a_t^{rs}; if yes, (ii) which is the best ad and (iii) where is the optimal location. Note that in this work we suppose that the AS can insert an ad into a given rec-list at most. Next, we discuss the limitations if we directly apply two classic Deep Q-network (DQN) architectures as shown in Figures 3.4(a) and (b) to the task. The architecture in Figure 3.4(a) inputs only the state (s_t and a_t^{rs}) and outputs Q-values of all $k+1$ possible locations. This DQN can select the optimal location, while it cannot choose the optimal ad to insert. The architecture in Figure 3.4(b) takes a pair of state-action and outputs the Q-value for a specific ad. This DQN can determine the optimal ad but cannot

determine where to insert the ad. One solution is to input the location information (e.g., one-hot vector). However, it needs $O(k|\mathcal{A}_t^{as}|)$ evaluations (forward propagation) to find the optimal Q-value, which is not practical in real-world advertising systems. Note that neither two classic DQNs can decide whether to insert an ad into a given rec-list.

To address above challenges, we propose a novel DQN architecture for online advertising in a given rec-list a_t^{rs}, as shown in Figure 3.4(c). It is built based on the two classic DQN architectures. The proposed DQN takes state s_t (including a_t^{rs}) and a candidate ad a_t^{ad} as input, and outputs the Q-values for all $k+1$ locations. This DQN architecture inherits the advantages of both two classic DQNs. It can evaluate the Q-values of the combination of two internally related types of sub-actions at the same time. In this chapter, we evaluate the Q-values of all possible locations for an ad simultaneously. To determine whether to insert an ad (the first sub-action), we extend the output layer from $k+1$ to $k+2$ units, where the $Q(s_t, a_t^{ad})^0$ unit corresponds to the Q-value of not inserting an ad into rec-list a_t^{rs}. Therefore, our proposed DQN can simultaneously determine three aforementioned sub-actions according to the Q-value of ad-location combinations (a_t^{ad}, a_t^{loc}), and the evaluation times are reduced from $O(k|\mathcal{A}_t^{as}|)$ to $O(|\mathcal{A}_t^{as}|)$; when $Q(s_t, \mathbf{0})^0$ leads to the maximal Q-value, the AS will insert no ad into rec-list a_t^{rs}, where we use a zero-vector $\mathbf{0}$ to represent inserting no ad.

More details of the proposed DQN architecture are illustrated in Figure 3.5. First, whether to insert an ad into a given rec-list is affected by s_t and a_t^{rs} (especially the quality of the rec-list). For example, if a user is satisfied with a rec-list, the AS may prefer to insert an ad into the rec-list; conversely, if a user is unsatisfied with a rec-list and is likely to leave, then the AS won't insert an ad. Second, the reward for an ad-location pair is related to all information. Thus, we divide the Q-function into a value function $V(s_t)$, related to s_t and a_t^{rs}, and an advantage function $A(s_t, a_t^{ad})$, decided by s_t, a_t^{rs} and a_t^{ad} [138].

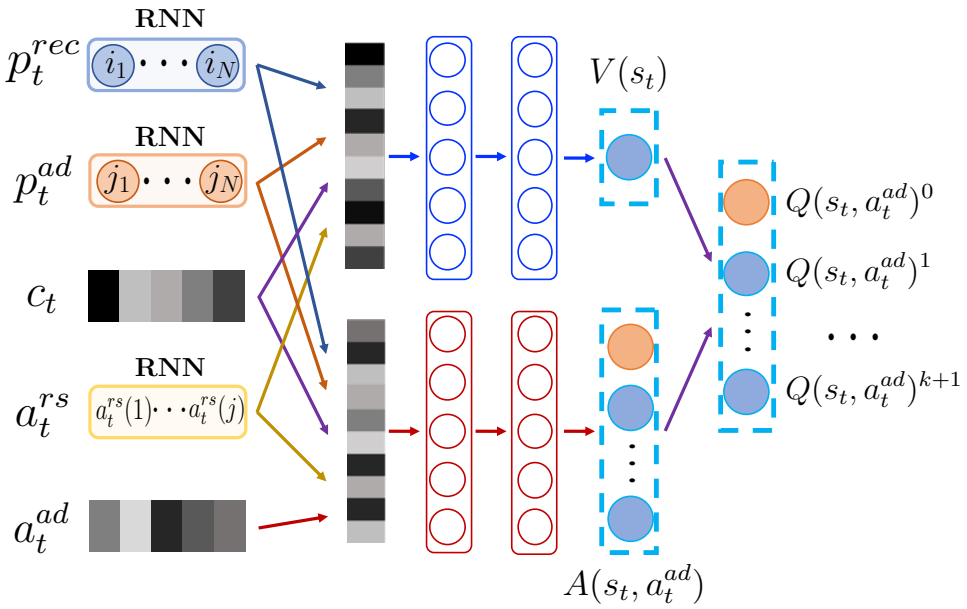

Figure 3.5: The architecture of the proposed DQN for AS.

3.3.2.3 The Action Selection in RTB setting

In the real-time bidding environment, each ad slot is bid by advertisers in real-time when an impression is just generated from a consumer visit [13]. In other words, given an ad slot, the specific ad to display is determined by the bids from advertisers, i.e., the bidding system (BS), rather than the platform, which aims to maximize the immediate advertising revenue of each ad slot from advertisers. In this chapter, as mentioned in Section 3.1, the optimal ad selection policy should not only maximize the immediate advertising revenue (controlled by the BS), but also minimize the negative influence of ads on user experience in the long run (controlled by the AS). To achieve this goal, the AS will first calculate the Q-values for all candidate ads and all possible locations, referred as to $Q(s_t, \mathcal{A}_t^{as})$, which captures the long-term influence of ads on user experience; and then the BS will select the ad that achieves the trade-off between the immediate advertising revenue and the long-term Q-values:

$$a_t^{as} = BS\left(Q(s_t, \mathcal{A}_t^{as})\right) \tag{3.7}$$

where the operation $Q(s_t, \mathcal{A}_t^{as})$ goes through all candidate ads $\{a_t^{ad}\}$ (input layer) and all locations $\{a_t^{loc}\}$ (output layer), including the location that represents not inserting an ad. To be more specific, we design two AS+BS approaches as follows:

- **RAM-l**: the optimal ad-location pair $a_t^{as} = (a_t^{ad}, a_t^{loc})$ directly optimizes the linear summation of immediate advertising revenue and long-term user experience:

$$a_t^{as} = \arg \max_{a_t^{as} \in \mathcal{A}_t^{as}} \left(Q(s_t, a_t^{as}) + \alpha \cdot rev_t(a_t^{as}) \right) \tag{3.8}$$

  where α controls the second term, and $rev_t(a_t^{as})$ is the immediate advertising revenue if inserting an ad, otherwise 0;

- **RAM-n**: this is a nonlinear approach that the AS first selects a subset of ad-location pairs $\{a_t^{as}\}$ (the size is N) that corresponds to optimal long-term user experience $Q(s_t, a_t^{as})$, then the BS chooses one a_t^{as} that has the maximal immediate advertising revenue $rev_t(a_t^{as})$ from the subset.

It is noteworthy that we maximize immediate advertising revenue rather than long-term advertising revenue because: (i) as aforementioned, advertisers determine the ad to insert rather than the platform (the agent does not generate action); and (ii) in the generalized-second-price (GSP) setting, the highest bidder pays the price (immediate advertising revenue) bid by the second-highest bidder, if we use immediate advertising revenue as $r_t(s_t, a_t^{as})$, then we cannot select an ad according to its $Q(s_t, a_t^{as})$ that represents the long-term advertising revenue.

3.3.3 The Optimization Task

Given a user request and state s_t, the RS and AS sequentially generate actions a_t^{rs} and a_t^{as}, i.e., a rec-ads hybrid list, and then the target user will browse the list and provide her/his feedback. The two-level framework aims to (i) optimize the long-term user experience or engagement of recommended items (RS), (ii) maximize the immediate advertising revenue from advertisers in RTB environment (BS), and (iii) minimize the negative influence of ads on user long-term experience (AS), where the second goal is automatically achieved by the bidding system, i.e., the advertiser with highest bid price will win the ad slot auction. Therefore, we next design proper reward functions to assist the RL components in the framework to achieve the first and third goals.

The framework is quite general for the rec-ads mixed display applications in e-commerce, news feed and video platforms. Thus, for different types of platforms, we design different reward functions. For the first level DQN (RS), to evaluate the user experience, we have

$$r_t(s_t, a_t^{rs}) = \begin{cases} income & e\text{-}commerce \\ dwell\,time & news/videos \end{cases} \tag{3.9}$$

where user experience is measured by the income of the recommended items in the hybrid list in e-commerce platforms, and the dwelling time of the recommendations in news/video platforms. Based on the reward function $r_t(s_t, a_t^{rs})$, we can update the parameters of the cascading DQN (RS) by performing gradient steps over the loss in Eq. (3.6). We introduce separated evaluation and target networks [93] to help smooth the learning and avoid the divergence of parameters, where θ^{rs} represents all parameters of the evaluation network, and the parameters of the target network θ_T^{rs} are fixed when optimizing the loss function. The

derivatives of loss function $L(\theta^{rs})$ with respective to parameters θ^{rs} are presented as follows:

$$\nabla_{\theta^{rs}} L(\theta^{rs}) = \left(y_t^{rs} - Q^j \left(s_t, a_t^{rs}(1{:}j); \theta^{rs} \right) \right) \nabla_{\theta^{rs}} Q^j \left(s_t, a_t^{rs}(1{:}j); \theta^{rs} \right) \tag{3.10}$$

where the target $y_t^{rs} = r_t \left(s_t, a_t^{rs}(1{:}k) \right) + \gamma Q^* \left(s_{t+1}, a_{t+1}^{rs*}(1{:}k); \theta_T^{rs} \right), \forall j \in [1, k]$.

For the second level DQN (AS), since leaving the platforms is the major risk of inserting ads improperly or too frequently, we evaluate user experience by whether user will leave the platform after browsing current rec-ads hybrid list, and we have:

$$r_t(s_t, a_t^{as}) = \begin{cases} 1 & continue \\ 0 & leave \end{cases} \tag{3.11}$$

in other words, the AS will receive a positive reward (e.g. 1) if the user continues to browse the next list, otherwise 0 reward. Then the optimal $Q^*(s_t, a_t^{as})$, i.e., the maximum expected return achieved by the optimal policy, follows the Bellman equation [6] as:

$$Q^*(s_t, a_t^{as}) = r_t(s_t, a_t^{as}) + \gamma Q^* \big(s_{t+1}, BS \left(Q^*(s_{t+1}, \mathcal{A}_{t+1}^{as}) \right) \big) \tag{3.12}$$

then the second level DQN can be optimized by minimizing the loss function as:

$$\begin{aligned} &\big(y_t^{as} - Q(s_t, a_t^{as}) \big)^2, \text{ where} \\ &y_t^{as} = r_t(s_t, a_t^{as}) + \gamma Q^* \big(s_{t+1}, BS \big(Q^*(s_{t+1}, \mathcal{A}_{t+1}^{as}) \big) \big) \end{aligned} \tag{3.13}$$

where y_t^{as} is the target of the current iteration. We also introduce separated evaluation and target networks [93] with parameters θ^{as} and θ_T^{as} for the second level DQN (AS), and θ_T^{as} is fixed when optimizing the loss function in Eq. (3.13) (i.e. $L(\theta^{as})$). The derivatives of loss

function $L(\theta^{as})$ w.r.t. parameters θ^{as} can be presented as:

$$\nabla_{\theta^{as}} L(\theta^{as}) = \big(y_t^{as} - Q(s_t, a_t^{as}; \theta^{as})\big) \nabla_{\theta^{as}} Q(s_t, a_t^{as}; \theta^{as}) \tag{3.14}$$

where $y_t^{as} = r_t(s_t, a_t^{as}) + \gamma Q^*\big(s_{t+1}, BS\big(Q^*(s_{t+1}, \mathcal{A}_{t+1}^{as}; \theta_T^{as})\big); \theta_T^{as}\big)$. The operation $Q(s_t, \mathcal{A}_t^{as})$ looks through the candidate ad set $\{a_{t+1}^{ad}\}$ and all locations $\{a_{t+1}^{loc}\}$ (including the location of inserting no ad).

3.3.4 Off-policy Training

Training the two-level reinforcement learning framework requires a large amount of user-system interaction data, which may result in bad user experience in the initial online stage when new algorithms have not been well trained. To address this challenge, we propose an off-policy training approach that effectively utilizes users' historical behavior log from other policies. The users' offline log records the interaction history between behavior policy $b(s_t)$ (the current recommendation and advertising strategies) and users' feedback. Our RS and AS take the actions based on the off-policy $b(s_t)$ and obtain feedback from the offline log. We present our off-policy training algorithm in detail shown in Algorithm 3.1.

There are two phases in each iteration of a training session. For the transition generation phase: for the state s_t (line 6), the RS and AS sequentially act a_t^{rs} and a_t^{as} based on the behavior policy $b(s_t)$ (line 7) according to a standard off-policy way [30]; then RS and AS receive the reward $r_t(s_t, a_t^{rs})$ and $r_t(s_t, a_t^{as})$ from the offline log (line 8) and update the state to s_{t+1} (line 9); and finally the RS and AS store transition $(s_t, a_t^{rs}, a_t^{as}, r_t(s_t, a_t^{rs}), r_t(s_t, a_t^{as}), s_{t+1})$ into the replay buffer $\mathcal{D}$ (line 10). For the model training phase: the proposed framework first samples minibatch of transitions from $\mathcal{D}$ (line 11), then generates actions $a^{rs'}$ and

Algorithm 3.1: Off-policy Training of the RAM Framework.

Input: historical offline logs, replay buffer $\mathcal{D}$

Output: well-trained recommending policy π^*_{rs} and advertising policy π^*_{as}

1: Initialize the capacity of replay buffer $\mathcal{D}$
2: Randomly initialize action-value functions Q_{rs} and Q_{as}
3: **for** session $= 1, M$ **do**
4: Initialize state s_0
5: **for** $t = 1, T$ **do**
6: Observe state $s_t = concat(p_t^{rec}, p_t^{asd}, c_t)$
7: Execute actions a_t^{rs} and a_t^{as} according to off-policy $b(s_t)$
8: Get rewards $r_t(s_t, a_t^{rs})$ and $r_t(s_t, a_t^{as})$ from offline log
9: Update the state from s_t to s_{t+1}
10: Store $(s_t, a_t^{rs}, a_t^{as}, r_t(s_t, a_t^{rs}), r_t(s_t, a_t^{as}), s_{t+1})$ transition into the $\mathcal{D}$
11: Sample minibatch of $(s, a^{rs}, a^{as}, r(s, a^{rs}), r(s, a^{as}), s')$ transitions from the $\mathcal{D}$
12: Generate RS's next action $a^{rs'}$ according to Eq. (3.4)
13: Generate AS's next action $a^{as'}$ according to Eq. (3.7)
14: $y^{rs} = \begin{cases} r(s, a^{rs}) & \text{terminal } s' \\ r(s, a^{rs}) + \gamma Q_{rs}(s', a^{rs'}) & \text{non} - \text{terminal } s' \end{cases}$
15: Update θ^{rs} of Q_{rs} by minimizing $\left(y^{rs} - Q^j_{rs}\left(s, a^{rs}(1:j)\right)\right)^2$ via Eq. (3.10)
16: $y^{as} = \begin{cases} r(s, a^{as}) & \text{terminal } s' \\ r(s, a^{as}) + \gamma Q_{as}(s', a^{as'}) & \text{non} - \text{terminal } s' \end{cases}$
17: Update θ^{as} of Q_{as} by minimizing $\left(y^{as} - Q_{as}(s, a^{as})\right)^2$ according to Eq. (3.14)
18: **end for**
19: **end for**

$a^{as'}$ of next iteration according to Eqs.(3.4) and (3.7) (lines 12-13), and finally updates parameters of Q_{rs} and Q_{as} by minimizing Eqs.(3.6) and .(3.13) (lines 14-17). To help avoid the divergence of parameters and smooth the training, we introduce separated evaluation and target Q-networks [93] . Note that when $b(s_t)$ decides not to insert an ad (line 7), we denote a_t^{ad} as an all-zero vector.

3.3.5 Online Test

We present the online test procedure in Algorithm 3.2. The process is similar to the transition generation stage of Algorithm 3.1. Next, we detail each iteration of test session as shown

Algorithm 3.2: Online Test of the RAM Framework.

1: Initialize action-value functions Q_{rs} and Q_{as} with well-trained weights
2: **for** session $= 1, M$ **do**
3: Initialize state s_0
4: **for** $t = 1, T$ **do**
5: Observe state $s_t = concat(p_t^{rec}, p_t^{asd}, c_t)$
6: Generate action a_t^{rs} according to Eq. (3.4)
7: Generate action a_t^{as} according to Eq. (3.7)
8: Execute actions a_t^{rs} and a_t^{as}
9: Observe rewards $r_t(s_t, a_t^{rs})$ and $r_t(s_t, a_t^{as})$ from user
10: Update the state from s_t to s_{t+1}
11: **end for**
12: **end for**

in Algorithm 3.2. First, the well-trained RS generates a rec-list by π_{rs}^* (line 6) according to the current state s_t (line 5). Second, the well-trained AS, collaboratively working with BS, decides to insert an ad into the rec-list (or not) by π_{as}^* (line 7). Third, the reward is observed from the target user to the hybrid list of recommended and advertised items (lines 8 and 9). Finally we transit the state to s_{t+1} (line 10).

3.4 Experiment

In this section, we will conduct extensive experiments using data from a short video site to assess the effectiveness of the proposed RAM framework. We first introduce the experimental settings, then compare the RAM framework with state-of-the-art baselines, and finally conduct component and parameter analysis on RAM.

3.4.1 Experimental Settings

Since there are no public datasets consist of both recommended and advertised items, we collected a dataset from a short video site, TikTok, in March 2019. In total, we collect

Table 3.1: Statistics of the dataset.

session	user	normal video	ad video
1,000,000	188,409	17,820,066	10,806,778
session dwell time	session length	session ad revenue	rec-list with ad
17.980 min	55.032 videos	0.667	55.23%

1,000,000 sessions in chronological order, where the first 70% is used as training/validation set and the later 30% is test set. For more statistics of the dataset, please see Table 3.1. There are two types of videos in the dataset: regular videos (recommended items) as well as ad videos (advertised items). The features for a normal video contain: id, like score, finish score, comment score, follow score and group score, where the scores are predicted by the platform. The features for an ad video consist of: id, image size, bid-price, hidden-cost, predicted-ctr and predicted-recall, where the platform predicts the last four. It is worth noting that (i) the effectiveness of the calculated features have been validated in the businesses of the short video site, (ii) we discretize each numeric feature into a one-hot vector, and (iii) baselines are based on the same features for a fair comparison.

3.4.2 Evaluation Metrics

The reward $r_t(s_t, a_t^{rs})$ to evaluate user experience of a list of regular videos is the dwell time (min), and the reward $r_t(s_t, a_t^{as})$ of ad videos is 0 if users leave the site and 1 if users continue to browse. We use the *session dwell time* $R^{rs} = \sum_1^T r_t(s_t, a_t^{rs})$, *session length* $R^{as} = \sum_1^T r_t(s_t, a_t^{as})$, and session ad revenue $R^{rev} = \sum_1^T rev_t(a_t^{as})$ as metrics to measure the performance of a test session.

3.4.3 Architecture Details

Next, we detail the architecture of RAM to ease reproductivity. The number of candidate regular/ad videos (selected by external recall systems) for a request is 15 and 5 respectively, and the size of regular/ad video representation is 60. There are $k = 6$ regular videos in a rec-list. The initial state s_0 of a session is collected from its first three requests, and the dimensions of $p_t^{rec}, p_t^{ad}, c_t, a_t^{rs}, a_t^{ad}$ are $64, 64, 13, 360, 60$, respectively. For the second level DQN (AS), two separate 2-layer neural networks are respectively used to generate $V(s_t)$ and $A(s_t, a_t^{ad})$, where the output layer has $k + 2 = 8$ units, i.e., 8 possible ad locations including not to insert an ad. We empirically set the size of replay buffer 10,000, and the discounted factor of MDP $\gamma = 0.95$. We select important hyper-parameters such as α and N via cross-validation, and we do parameter-tuning for baselines for a fair comparison. In the following subsections, we will present more details of parameter sensitivity for the RAM framework.

3.4.4 Overall Performance Comparison

The experiment is based on a simulated online environment, which can provide the $r_t(s_t, a_t^{rs})$, $r_t(s_t, a_t^{as})$ and $rev_t(a_t^{as})$ according to a mixed rec-ads list. The simulator shares similar architecture to Figure 3.5, while the output layer predicts the dwell time, whether user will leave and the ad revenue of current mixed rec-ads list. We compare the proposed framework with the following representative baseline methods:

- **W&D** [24]: This baseline jointly trains a wide linear model with feature transformations and a deep feedforward neural network with embeddings for general recommender systems with sparse inputs. One W&D estimates the recommending scores of regular videos and

Table 3.2: Performance comparison.

Metrics	Values	Algorithms					
		W&D	DFM	GRU	DRQN	RAM-l	RAM-n
R^{rs}	value	17.61	17.95	18.56	18.99	**19.61**	19.49
	improv.(%)	11.35	9.25	5.66	3.26	-	0.61
	p-value	0.000	0.000	0.000	0.000	-	0.006
R^{as}	value	8.79	8.90	9.29	9.37	**9.76**	9.68
	improv.(%)	11.03	9.66	5.06	4.16	-	0.83
	p-value	0.000	0.000	0.000	0.000	-	0.009
R^{rev}	value	1.07	1.13	1.23	1.34	1.49	**1.56**
	improv.(%)	45.81	38.05	26.83	16.42	4.70	-
	p-value	0.000	0.000	0.000	0.000	0.001	-

each time we recommend k videos with highest scores, while another W&D predicts whether to insert an ad and estimates the CTR of ads.

- **DFM** [51]: DeepFM is a deep model that incorporates W&D model with factorization-machine (FM). It models high-order feature interactions like W&D and low-order interactions like FM.

- **GRU** [55]: GRU4Rec is an RNN with GRU to predict what user will click next according to her/his behavior histories.

- **DRQN** [53]: Deep Recurrent Q-Networks addresses the partial observation problem by considering the previous context with a recurrent structure. DRQN uses an RNN architecture to encode previous observations before the current time.

Note that we also develop two separate DFMs, GRUs, DRQNs for RS and AS, respectively. The results are shown in Table 3.2. We make the following observations:

- GRU performs better than W&D and DFM, since W&D and DFM neglect users' sequential behaviors of one session, while GRU can capture the sequential patterns.

- DRQN outperforms GRU, since DRQN aims to maximize the long-term rewards of a session, while GRU targets maximizing the immediate reward of each request. This result demonstrates the advantage of introducing RL for online recommendation and advertising.
- RAM-l and RAM-n achieve better performance than DRQN, which validates the effectiveness of the proposed two-level DQN framework, where the RS generates a rec-list of recommendations and the AS decides how to insert ads.
- RAM-n outperforms RAM-l in session ad revenue, since the second step of RAM-n will select the ad-location pair with maximal immediate advertising revenue, which has a higher probability of inserting ads.

To sum up, RAM outperforms representative baselines, which demonstrates its effectiveness in online recommendation and advertising.

3.4.5 Component Study

To understand the impact of model components of RAM, we systematically eliminate the corresponding components of RAM by defining the following variants:

- **RAM-1**: This variant has the same neural network architectures with the RAM framework, while we train it in the supervised learning way;
- **RAM-2**: In this variant, we evaluate the contribution of recurrent neural networks, so we replace RNNs with fully-connected layers. Specifically, we concatenate recommendations or ads into one vector and then feed it into fully-connected layers;
- **RAM-3**: In this variant, we use the original cascading DQN architecture in [23] as RS;

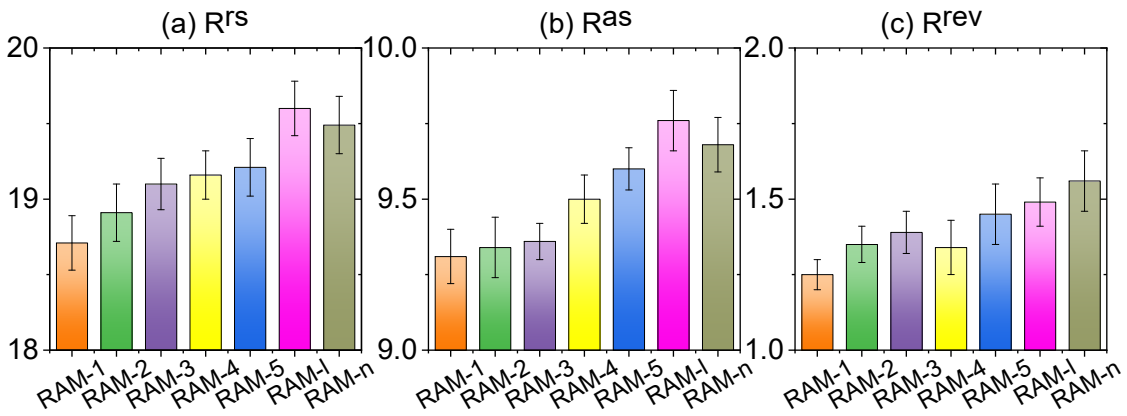

Figure 3.6: Performance comparison of different variants.

- **RAM-4**: For this variant, we do not divide the Q-function of AS into the value function $V(s)$ and the advantage function $A(s, a)$;
- **RAM-5**: This variant leverages an additional input to represents the location, and uses the DQN in Figure 3.4(b) for AS.

The results are shown in Figure 3.6. By comparing RAM and its variants, we make the following observations:

- RAM-1 demonstrates the advantage of reinforcement learning over supervised learning for jointly optimizing recommendation and online advertising;
- RAM-2 validates that capturing user's sequential behaviors can enhance the performance;
- RAM-3 proves the effectiveness of RNN over k separate layers for larger action space;
- RAM-4 suggests that dividing $Q(s_t, a_t)$ into $V(s_t)$ and $A(s_t, a_t)$ can boost the performance;
- RAM-5 validates the advantage of the proposed AS architecture (over classic DQN architectures) that inputs a candidate ad a_t^{ad} and outputs the Q-value for all possible locations $\{a_t^{loc}\}$.

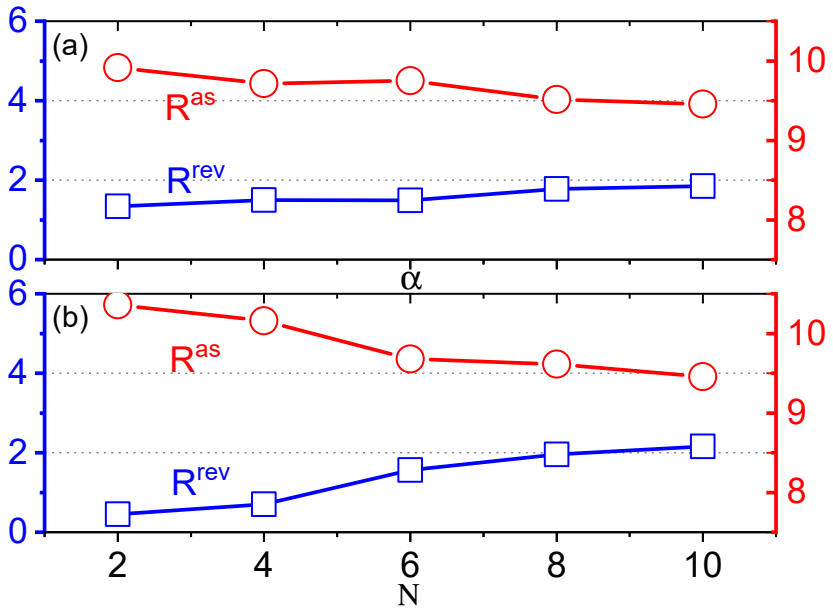

Figure 3.7: Parameter sensitivity analysis.

In summary, leveraging suitable RL policy and proper neural network components can improve the overall performance.

3.4.6 Parameter Sensitivity Analysis

Our method has two key hyper-parameters, i.e., (i) the parameter α of RAM-l, and (ii) the parameter N of RAM-n. To study their sensitivities, we fix other parameters, and investigate how the RAM framework performs with the changes of α or N.

- Figure 3.7(a) illustrates the sensitivity of α. We observe that when α increases, the metric R^{rev} improves, while the metric R^{as} decreases. This observation is reasonable because when we decrease the importance of the second term of Eq. (3.8), the AS will insert more ads or choose the ads likely to have more revenue, while ignoring their negative impact on regular recommendations.

- Figure 3.7(b) shows the sensitivity of N. With the increase of N, we can observe that

the metric R^{rev} improves and the metric R^{as} decreases. With smaller N, the first step of RAM-n prefers to selecting most ad-location pairs that do not insert an ad, which results in lower R^{rev} and larger R^{as}; on the contrary, with larger N, the first step returns more pairs with non-zero ad revenue, then the second step leads to higher R^{rev}.

In a nutshell, both above results demonstrate that recommended and advertised items are mutually influenced: inserting more ads can lead to more ad revenue while worse user experience, vice versa. Therefore, online platforms should carefully select these hyper-parameters according to their business demands.

3.5 Related Work

In this section, we will briefly summarize the related works of our study, which can be mainly grouped into the following categories.

The first category related to this chapter is reinforcement learning-based recommender systems. A DDPG algorithm is used to mitigate the large action space problem in real-world RL-based RS [32]. A tree-structured policy gradient is presented in [18] to avoid the inconsistency of DDPG-based RS. Biclustering is also used to model RS as grid-world games to reduce action space [26]. A Double DQN-based approximate regretted reward technique is presented to address the issue of unstable reward distribution in dynamic RS environment [21]. A pairwise RL-based RS framework is proposed to capture users' positive and negative feedback to improve recommendation performance [164]. A page-wise RS is proposed to simultaneously recommend a set of items and display them in a 2-dimensional page [160, 165]. A DQN based framework is proposed to address the issues in the news feed scenario, like only optimizing current reward, not considering labels, and diversity issue [167].

An RL-based explainable RS is presented to explain recommendations and can flexibly control the explanation quality according to the scenarios [136]. A policy gradient-based RS for YouTube is proposed to address the biases in logged data by introducing a simulated historical policy and a novel top-K off-policy correction [20].

The second category related to this chapter is RL-based online advertising techniques, which belong to two groups. The first group is guaranteed delivery (GD), where ads are charged according to a pay-per-campaign pre-specified number of deliveries [113]. A multi-agent RL method is presented to control cooperative policies for the publishers to optimize their targets in a dynamic environment [140]. The second group is real-time bidding (RTB), which allows an advertiser to bid each ad impression in a very short time slot. Ad selection task is typically modeled as multi-armed bandit problem supposing that arms are iid, feedback is immediate and environments are stationary [98, 41, 129, 148, 152, 114]. The problem of online advertising with budget constraints and variable costs is studied in MAB setting [31], where pulling the arms of bandit results in random rewards and spends random costs. However, the MAB setting considers the bid decision as a static optimization problem, and the bidding for a given ad campaign would repeatedly happen until the budget runs out. To address these challenges, the MDP setting has also been studied for RTB [13, 134, 160, 109, 141, 61]. A model-based RL framework is proposed to learn bid strategies in RTB setting [13], where state value is approximated by a neural network to better handle the large scale auction volume problem and limited budget. A model-free RL method is also designed to solve the constrained budget bidding problem, where a RewardNet is presented to generate rewards for reward design trap [141]. A multi-agent RL framework is presented to consider other advertisers' bidding as the state, and a clustering method is leveraged to handle a large amount of advertisers issue [61].

Chapter 4

User Simulation for Recommendations

Abstract

With the recent advances in Reinforcement Learning (RL), there have been tremendous interests in employing RL for recommender systems. However, directly training and evaluating a new RL-based recommendation algorithm needs to collect users' real-time feedback in the real system, which is time/effort consuming and could negatively impact users' experiences. Thus, it calls for a user simulator that can mimic real users' behaviors to pre-train and evaluate new recommendation algorithms. Simulating users' behaviors in a dynamic system faces immense challenges – (i) the underlying item distribution is complex, and (ii) historical logs for each user are limited. In this chapter, we develop a user simulator based on a Generative Adversarial Network (GAN). To be specific, the generator captures the underlying distribution of users' historical logs and generates realistic logs that can be considered as augmentations of real logs; while the discriminator not only distinguishes real and fake logs but also predicts users' behaviors. The experimental results based on benchmark datasets demonstrate the effectiveness of the proposed simulator.

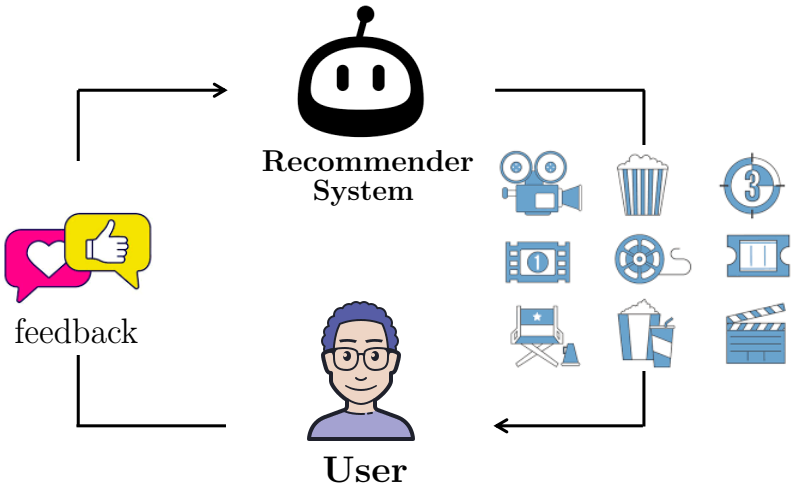

Figure 4.1: An example of system-user interactions.

4.1 Introduction

With the recent tremendous development in Reinforcement Learning (RL), there have been increasing interests in adapting RL for recommendations [20]. RL-based recommender systems treat the recommendation procedures as sequential interactions between users and a recommender agent (RA) as shown in Figure 4.1. In each iteration, the recommender system suggests a set of items to the user; then, the user browses the recommended items and provides her/his real-time feedback; next, the system will update its recommendation strategy according to user's feedback. RL-based recommender systems aim to automatically learn an optimal recommendation strategy (policy) that maximizes cumulative rewards from users without any specific instructions. They can achieve two key advantages: (i) the recommender agent can learn their recommendation strategies based on users' real-time feedback during the user-agent interactions continuously; and (ii) the optimal strategy targets at maximizing the long-term reward from users (e.g., the overall revenue of a recommendation session). Given the advantages of reinforcement learning, very recently, it allures tremendous interest in developing RL-based recommender systems. [32, 160, 167].

RL-based recommendation algorithms are desired to be trained and evaluated based on users' real-time feedback (reward function). The most practical and precise way is online A/B test [151, 66], where a new recommendation algorithm is trained based on the feedback from real users and the performance is compared against that of the previous algorithm via randomized experiments. However, online A/B tests are inefficient and expensive: (i) online A/B tests usually take several weeks to collect sufficient data for the sake of statistical sufficiency, and (ii) numerous engineering efforts are typically required to deploy the new algorithm in the real system [150, 43, 73]. Furthermore, online A/B tests often lead to bad user experience in the initial stage when the new recommendation algorithms have not been well trained [72]. These reasons prevent us from quickly training and testing new RL-based recommendation algorithms. The common practice to handle these challenges in the RL community is to build a simulator to approximate the environment (e.g., OpenAI Gym for video games), and then use it to train and evaluate the RL algorithms [39]. Thus, following the best routine, we aim to build a user simulator based on users' historical logs in this work, which can be utilized to pre-train and evaluate new recommendation algorithms before launching them online.

However, simulating users' behaviors in a dynamic recommendation environment is very challenging. First, the underlying distribution of recommended item sequences is extremely complex in historical logs since there are millions of items in practical recommender systems. Second, learning a robust simulator typically requires large-scale historical logs as training data from each user. Though massive historical logs are often available, data available to each user is rather limited. Recent efforts have demonstrated that Generative Adversarial Network (GAN) and its variants are able to generate fake but realistic images [45, 46], which implies their potential in modeling complex distributions. Furthermore, the generated images

can be considered as augmentations of real images to enlarge the data space. Driven by these advantages, we propose to build a GAN-based user simulator (UserSim) for RL-based recommenders, which can capture the complex distribution of users' browsing logs and generate realistic logs to enrich the training dataset. We summarize our major contributions as follows:

- We introduce a principled approach to capture the underlying distribution of recommended item sequences in historical logs, and generate realistic item sequences;
- We propose a user behavior simulator UserSim, which can be utilized to simulate environments with limited training data to pre-train and evaluate RL based recommender systems; and
- We conduct experiments based on real-world data to demonstrate the effectiveness of the proposed simulator and validate the contributions of its components.

4.2 The Proposed Simulator

This section will propose a simulator framework that imitates users' feedback (behavior) on a recommended item according to the user's current preference learned from her browsing history.

4.2.1 Problem Statement

RL-based recommender systems treat the recommendation task as sequential interactions between a recommender system (agent) and users (environment $\mathcal{E}$), and use a Markov Decision Process (MDP) to model them, which consist of a sequence of states, actions and rewards:

- We define the state $s = \{i_1, \cdots, i_N\}$ as a sequence of N items that a user browsed and user's corresponding feedback for each item. The items in s are chronologically sorted;
- An action a from the recommender system perspective is defined as recommending a set of items to the user. Without loss of generality, we suppose that each time the recommender system suggests one item to the user, but it is straightforward to extend this setting to recommending more items;
- When the system takes an action a based on the state s, the user will browse the recommended item and provide her feedback on the item, such as skip, click, or purchase the item. The recommender system will then receive a reward $r(s, a)$ solely according to the type of feedback.

With the aforementioned definitions and notations, the goal of a simulator can be formally defined as follows: *Given a state-action pair (s, a), the goal is to imitate user's feedback (behavior) on a recommended item according to user's preference learned from the user's browsing history.*

4.2.2 The Generator Architecture

The goal of the generator is to learn the data distribution and then generate indistinguishable logs (action) based on users' browsing history (state), i.e., to imitate the recommendation policy of the recommender system that generates the historical logs. Figure 4.2 illustrates the generator with the Encoder-Decoder architecture.

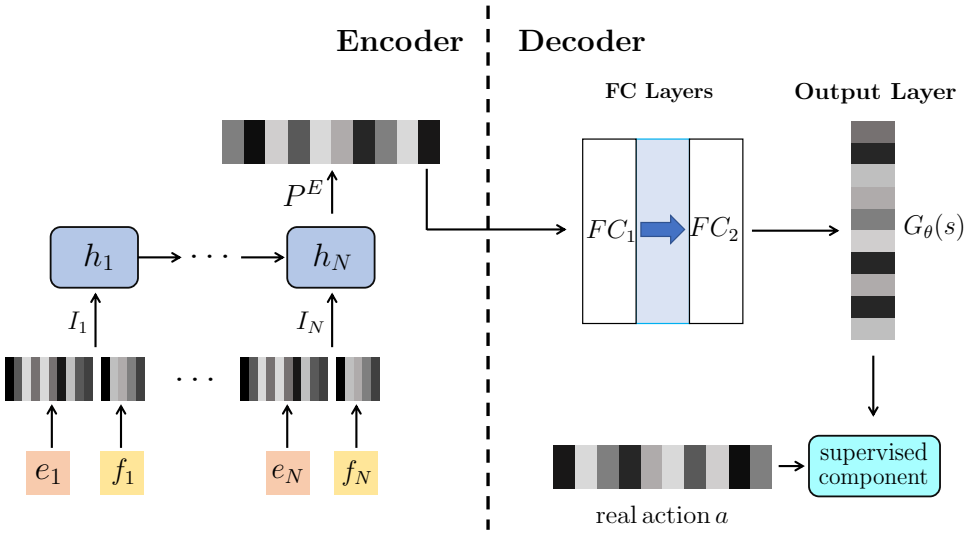

Figure 4.2: The generator with Encoder-Decoder architecture.

4.2.2.1 The Encoder Component

The Encoder component aims to learn user's preference according to the items browsed by the user and the user's feedback. The input is the state $s = \{i_1, \cdots, i_N\}$ that is observed in the historical logs, i.e., the sequence of N items that a user browsed and user's corresponding feedback for each item. The output is a low-dimensional representation of user's current preference, referred to as $\boldsymbol{p^E}$. Each item $i_n \in s$ involves two types of information: $i_n = (\boldsymbol{e}_n, \boldsymbol{f}_n)$, where $\boldsymbol{e}_n$ is a low-dimensional and dense item-embedding of the recommended item, and $\boldsymbol{f}_n$ is an embedding to denote user's feedback on the recommended item[1]. The intuition of selecting these two types of information is that, we not only want to learn the information of each item in the sequence, but also want to capture user's interests (feedback) on each item. Then, we concatenate $\boldsymbol{e}_n$ and $\boldsymbol{f}_n$, and get a low-dimensional and dense vector: $\boldsymbol{I}_n = concat(\boldsymbol{e}_n, \boldsymbol{f}_n)$.

We introduce a Recurrent Neural Network (RNN) with Gated Recurrent Units (GRU) to capture the sequential patterns of items in the logs. We consider the RNN's final hidden

[1]These embeddings are jointly trained with neural networks in an end-to-end manner.

state $\boldsymbol{h}_N$ as the output of Encoder component, i.e., the lower dimensional representation of user's current preference: $\boldsymbol{p}^{\boldsymbol{E}} = \boldsymbol{h}_N$.

4.2.2.2 The Decoder Component

The goal of the Decoder component is to predict the item that will be recommended according to the user's current preference. Therefore, the input is user's preference representation $\boldsymbol{p}^{\boldsymbol{E}}$, while the output is the item-embedding of the item that is predicted to appear at next position in the log, referred to as $G_\theta(s)$. We leverage a MLP component with several fully-connected layers as the Decoder to directly transform $\boldsymbol{p}^{\boldsymbol{E}}$ to $G_\theta(s)$. So far, we have delineated the architecture of the Generator, which aims to imitate the recommendation policy of the existing recommender system, and generate realistic logs to augment the historical data. In addition, we add a supervised component to encourage the generator to yield items that are close to the ground truth items, which will be discussed in Section 4.2.4. Next, we will discuss the architecture of discriminator.

4.2.3 The Discriminator Architecture

The discriminator aims to not only distinguish real historical logs and generated logs, but also predict the class of user's feedback on a recommended item according to her browsing history. Thus we consider the problem as a classification problem with $2 \times K$ classes, i.e., K classes of *real* feedback for the recommended items observed from historical logs, and K classes of *fake* feedback for the recommended items yielded by the generator.

Figure 4.3 illustrates the architecture of the discriminator. Similar with the generator, we introduce an RNN with GRU to capture user's dynamic preference. Note that the architecture is the same as the RNN in generator, but they have separate parameters.

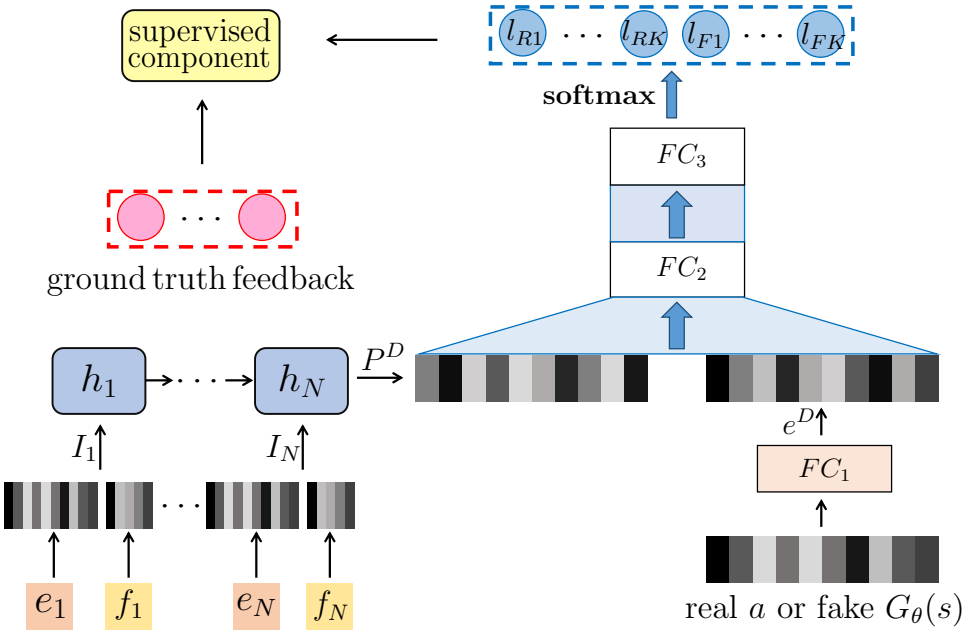

Figure 4.3: The discriminator architecture.

The input of the RNN is the state $s = \{i_1, \cdots, i_N\}$ observed in the historical logs, where $i_n = (\boldsymbol{e}_n, \boldsymbol{f}_n)$, and the output is the dense representation of the user's current preference, referred to as $\boldsymbol{p^D}$. Meanwhile, we feed the item-embedding of the recommended item (real a or fake $G_\theta(s)$) into fully-connected layers, which encode the recommended items to low-dimensional representations, referred to as $\boldsymbol{e^D}$. Then we concatenate $\boldsymbol{p^D}$ and $\boldsymbol{e^D}$, and feed the concatenation $(\boldsymbol{p^D}, \boldsymbol{e^D})$ into fully-connected layers, whose goals are (1) to judge whether the recommended items are real or fake, and (2) to predict users' feedback on these items. Therefore, the final fully-connected layer outputs a $2 \times K$ dimensional vector of logits, which represent K classes of *real* feedback and K classes of *fake* feedback respectively:

$$output = [l_{R1}, \cdots, l_{RK}, l_{F1}, \cdots, l_{FK}] \tag{4.1}$$

where we include K classes of *fake* feedback in output layer rather than only one *fake* class, since fine-grained distinction on fake samples can increase the power of discriminator (more

details in following subsections). These logits can be transformed to class probabilities through a softmax layer, and the probability corresponding to the j^{th} class is:

$$p_{model}(l_j|s,a) = \frac{exp(l_j)}{\sum_{k=1}^{2\times K} exp(l_k)} \tag{4.2}$$

The objective function is based on these class probabilities. In addition, a supervised component is introduced to enhance the user's feedback prediction and more details about this component will be discussed in Section 4.2.4.

4.2.4 The Objective Function

In this subsection, we will introduce the objective functions of the proposed simulator. The discriminator has two goals: (1) distinguishing real-world historical logs and generated logs, and (2) predicting the class of user's feedback on a recommended item according to the browsing history. The first goal corresponds to an unsupervised problem just like standard GAN that distinguishes real and fake images, while the second goal is a supervised problem that minimizes the class difference between users' ground truth feedback and the predicted feedback. Therefore, the loss function L_D of discriminator consists of two components.

For the unsupervised component that distinguishes real-world historical logs and generated logs, we need to calculate the probability that a state-action pair is *real* or *fake*. From Eq. (4.2), we know the probability that a state-action pair observed from historical logs is classified as *real*, referred to as $D_\phi(s,a)$, is the summation of the probabilities of K *real* feedback:

$$D_\phi(s,a) = \sum_{k=1}^{K} p_{model}(l_k|s,a) \tag{4.3}$$

while the probability of a *fake* state-action pair where $G_\theta(s)$ action is produced by the generator, say $D_\phi(s, G_\theta(s))$, is the summation of the probabilities of K *fake* feedback:

$$D_\phi(s, G_\theta(s)) = \sum_{k=K+1}^{2\times K} p_{model}(l_k|s, G_\theta(s)) \tag{4.4}$$

Then, the unsupervised component of the loss function L_D is defined as follows:

$$L_D^{unsup} = -\{\mathbb{E}_{s,a\sim p_{data}} \log D_\phi(s, a) + \mathbb{E}_{s\sim p_{data}} \log D_\phi(s, G_\theta(s))\} \tag{4.5}$$

where both s and a are sampled from historical logs distribution p_{data} in the first term; in the second term, only s is sampled from historical logs distribution p_{data}, while the action $G_\theta(s)$ is yielded by generator policy G_θ.

The supervised component aims to predict the class of user's feedback, which is formulated as a supervised problem to minimize the class difference (i.e., the cross-entropy loss) between users' ground truth feedback and the predicted feedback. Thus it also has two terms – the first term is the cross-entropy loss between ground truth class l_k and predicted class for a real state-action pair sampled from real historical data distribution p_{data}; while the second term is the cross-entropy loss between ground truth class l_k and the predicted class for a fake state-action pair, where the action $G_\theta(s)$ is yielded by the generator. Thus, the supervised component of the loss function L_D is defined as follows:

$$\begin{aligned} L_D^{sup} = -\{&\mathbb{E}_{s,a,r\sim p_{data}}[\log p_{model}(l_k|s, a, k{\leq}K)] \\ &+ \lambda \cdot \mathbb{E}_{s,r\sim p_{data}}[\log p_{model}(l_k|s, G_\theta(s), K{<}k{\leq}2K\} \end{aligned} \tag{4.6}$$

where λ controls the contribution of the second term. The first term is a standard cross

entropy loss of a supervised problem. The intuition we introduce the second term of Eq. (4.6) is – in order to tackle the data limitation challenge mentioned in Section 4.1, we consider fake state-action pairs as augmentations of real state-action pairs. Then fine-grained distinction on fake state-action pairs will increase the power of discriminator, which also in turn forces the generator to output more indistinguishable actions. In other words, user will provide the same feedback if the generated item is sufficiently similar to the real one. The overall loss function of the discriminator L_D is defined as follows:

$$L_D = L_D^{unsup} + \alpha \cdot L_D^{sup} \tag{4.7}$$

where parameter α is introduced to control the contribution of the supervised component.

The target of the generator is to output realistic recommended items $G_\theta(s)$ that can fool the discriminator, which tackles the complex data distribution problem as mentioned in Section 4.1. To achieve this goal, we design two components for the loss function L_G of the generator. The first component aims to maximize L_D^{unsup} in Eq. (4.5) with respect to G_θ. In other words, the first component minimizes that probabilities that fake state-action pairs are classified as *fake*, thus we have:

$$L_G^{unsup} = \mathbb{E}_{s \sim p_{data}}[\log D_\phi(s, G_\theta(s))] \tag{4.8}$$

where s is sampled from real historical logs distribution p_{data} and the action $G_\theta(s)$ is yielded by generator policy G_θ. Inspired by a supervised version of GAN [86], we introduce a supervised loss L_G^{sup} as the second component of L_G, which is the ℓ_2 distance between the

Algorithm 4.1: Training Algorithm for the Simulator.

1: Initialize the generator G_θ and discriminator D_ϕ with random weights θ and ϕ
2: Sample a pre-training dataset of $s, a \sim p_{data}$
3: Pre-train G_θ by minimizing L_G^{sup} in Eq. (4.9)
4: Generate fake-actions $G_\theta(s) \sim G_\theta$ for training D_ϕ
5: Pre-train D_ϕ by minimizing L_D^{sup} in Eq. (4.6)
6: **repeat**
7: **for** d-steps **do**
8: Sample minibatch of $s, a \sim p_{data}$
9: Use current G_θ to generate minibatch of $G_\theta(s) \sim G_\theta$
10: Update the D_ϕ by minimizing L_D in Eq. (4.7)
11: **end for**
12: **for** g-steps **do**
13: Sample minibatch of $s, a \sim p_{data}$
14: Update the G_θ by minimizing L_G in Eq. (4.10)
15: **end for**
16: **until** simulator converges

ground truth item a and the generated item $G_\theta(s)$:

$$L_G^{sup} = \mathbb{E}_{s,a\sim p_{data}} \|a - G_\theta(s)\|_2^2 \tag{4.9}$$

where s and a are sampled from historical logs distribution p_{data}. This supervised component encourages the generator to yield items that are close to the ground truth items. The overall loss function of the generator L_G is defined as follows:

$$L_G = L_G^{unsup} + \beta \cdot L_G^{sup} \tag{4.10}$$

where β controls the contribution of the second component.

We detail our simulator training algorithm in Algorithm 4.1. At the beginning of the training stage, we use standard supervised methods to pre-train the generator (line 3) and discriminator (line 5). After the pre-training stage, discriminator (lines 7-11) and generator

Table 4.1: Statistics of the datasets.

Dataset	user (session)	item	interaction	ave. length
JD.com	283,228	1,355,255	97,713,660	345

(lines 12-15) are trained alternatively. For training the discriminator, state s and real action a are sampled from real historical logs, while fake actions $G_\theta(s)$ are generated through the generator. To keep balance in each d-step, we generate fake actions $G_\theta(s)$ with the same number of real actions a.

4.3 Experiments

In this section, we conduct extensive experiments to evaluate the effectiveness of the proposed simulator on real-world datasets.

4.3.1 Experimental Settings

We evaluate our method on public JD.com dataset[2]. The statistics are shown in Table 4.1. We consider the whole sequence of item-feedback pairs of each user as a session, and consider *click* as positive and *skip* as negative. For each session, we use first $N = 20$ items and corresponding feedback as the initial state, the $N + 1^{th}$ item as the first action, then we could collect a sequence of (state,action,reward) tuples following the MDP defined in Section 4.2.1. We collect the last 30% (state,action,reward) tuples from each session as the test set, while using the previous 70% tuples as the training/validation set.

In our experiments, we leverage $N = 20$ items that a user browsed and user's corresponding feedback for each item as state s. The dimension of the item-embedding e_n is $|E| = 35$,

[2]https://datascience.jd.com/page/opendataset.html

and the dimension of feedback-embedding f_n is $|F| = 15$. The output of discriminator is a 4 (*i.e.*, $K = 2$) dimensional vector of logits, and each logit represents *real-positive*, *real-negative*, *fake-positive* and *fake-negative* respectively:

$$output = [l_{rp}, l_{rn}, l_{fp}, l_{fn}] \tag{4.11}$$

where *real* denotes that the recommended item is observed from historical logs; *fake* denotes that the recommended item is yielded by the generator; *positive* denotes the positive feedback such as a user clicks/purchases the recommended item; and *negative* denotes the negative feedback such as a user skips the recommended item. Note that though we only simulate two types of behaviors of users (i.e., positive and negative), it is straightforward to extend the simulators with more types of behaviors (e.g., purchase and leave). AdamOptimizer is used for optimization, and the learning rate for both Generator and Discriminator is 0.001, and batch-size is 500. The hidden size of RNN is 128. For the hyper-parameters of we use in the proposed framework such as $N = 20$, $\lambda = 0.3$, $\alpha = 0.7$ and $\beta = 0.7$, we select them via cross-validation. Correspondingly, we also do parameter-tuning for baselines for a fair comparison.

In the test stage, given a state-action pair, the simulator will predict the classes of user's feedback for the action (recommended item), and then compare the prediction with ground truth feedback observed from the historical log. For this classification task, we select the commonly used *F1-score* [101] as the metric, which is a measure that combines precision and recall, namely the harmonic mean of precision and recall. Moreover, we leverage $p_{model}(l_{rp}|s, a)$ (i.e. the probability that user will provide positive feedback to a real recommended item) as the score, and use AUC (Area under the ROC Curve) [27] as the

metric to evaluate the performance.

4.3.2 Overall Performance

We compare the proposed model with the following state-of-the-art baseline methods:

- **LR**: Logistic Regression [92] uses a logistic function to model a binary dependent variable through minimizing the loss $\mathbb{E}\frac{1}{2}(h_\theta(x) - y)^2$, where $h_\theta(x) = \frac{1}{1+e^{-w^T x}}$; we concatenate all $i_n = (e_n, f_n)$ as the feature vector for the i-th item, and set ground truth $y = 1$ if feedback is positive, otherwise $y = 0$.

- **UserSim-d**: This baseline has a similar architecture with the proposed discriminator. The differences are: we feed real recommended items as input action rather than both real and fake items; and in output layer, we predict the class of user's feedback to this real item without considering the fake feedback.

- **RecSim**: RecSim [56] is a configurable platform for authoring simulation environments for recommender systems via a dynamic Bayesian network that defines a probability distribution over trajectories of items, choices, and observations.

- **RecoGym**: RecoGym [109] a bandit-based recommender system simulation environment that combine organic navigation with intermittent recommendation (or ads).

- **VT**: Virtual-taobao [117] is a user estimation model that generates virtual interactions through multi-agent adversarial imitation learning. It has two independent GANs: one for generating customer features and the other for generating interactions.

- **GAN-PW**: GAN user model with position weight [23] imitates users' sequential choices by imitation learning, which formulates a unified mini-max optimization to learn user

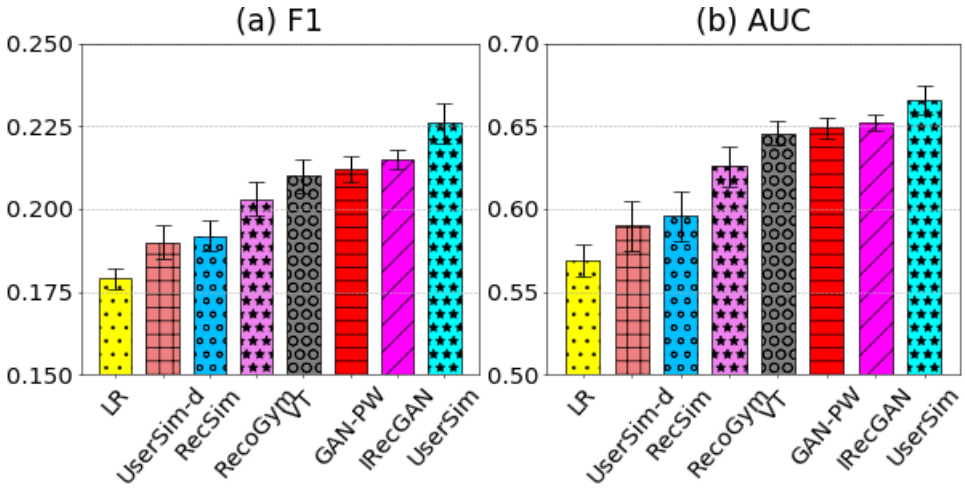

Figure 4.4: The results of overall performance comparison.

behavior model and reward function simultaneously based on sample trajectories.

- **IRecGAN**: the user model in [4] interacts with recommender agent to generate recommendation sequences that are close to the true data distribution via a generative adversarial network.

The overall performances of UserSim (discriminator) and baselines are shown in Figure 4.4. We make the following observations:

- UserSim-d and RecSim outperform LR, since LR neglects the temporal sequence within users' browsing history, while UserSim-d and RecSim can capture the temporal patterns within the item sequences and users' feedback for each item. This result demonstrates that it is important to capture the sequential patterns of users' browsing history when learning users' preferences.

- RecoGym achieves worse performance than VT and UserSim, since RecoGym is a bandit-based model that does not allow user state transitions. Furthermore, VT and UserSim are GAN-based models that are efficient at capturing the underlying distribution of item sequences.

- UserSim outperforms UserSim-d, which shares a similar architecture with the proposed discriminator, but lacks the generator component. This observation validates that the generated logs can actually lead to improvements in feedback predictions.

- UserSim performs better than VT, because VT is built upon two independent GANs trained separately, while UserSim jointly learns the sequence generation and user feedback prediction into one unified GAN framework. Also, UserSim takes advantage of both the unsupervised and supervised components, while VT consists of only unsupervised ones.

- UserSim gets better than performance GAN-PW, the reasons involve: (i) UserSim leverages RNN with GRU to capture user's preference from browsing history, while GAN-PW uses positional weight separately without efficient weight sharing [118]; (ii) UserSim's supervision signal enhances its performance.

- UserSim outperforms IRecGAN, where the user model and next fake item producer are both involved in IRecGAN's generator, and its discriminator only distinguishes real and fake items. This design breaks the naturally adversarial relationship between user model and next fake item producer (i.e., the producer aims to generate near-real items to fool user model, while user model targets distinguishing real and fake items), which leads to suboptimal performance.

To sum up, the proposed framework outperforms the state-of-the-art baselines with significant margin, which validates its effectiveness in simulating users' behaviors in recommendation tasks.

Figure 4.5: The training process of RL-based recommenders.

4.3.3 RL-based Recommender Training

In this subsection, we evaluate the effectiveness of the proposed simulator on training reinforcement learning based recommender systems. Since real online environment is not available, we train a recommender based on a deep Q-network (DQN) framework. This off-policy reinforcement learning method can train on historical offline user behavior logs. We compare the training process of three recommenders with the same architecture, where the first is directly trained based on historical offline logs, the second is trained based on IRecGAN (the best baseline in Section 4.3.2), and the third is trained based on UserSim. Note that both IRecGAN and UserSim are learned upon the same historical offline logs. We use the averaged rewards on the test set (say *avg_reward*) as metric to evaluate the performance of recommenders. Figure 4.5 illustrates the training process of the recommenders. We can observe that:

- In the initial training stage, the *avg_reward* of all recommenders grow rapidly, then their growth speed gradually becomes slower with more interaction data.

- When the recommenders achieve convergence, the recommender trained upon UserSim converges to the similar *avg_reward* value with the one trained upon offline logs, while IRecGAN's recommender converges to a distinctly different *avg_reward*.

- The recommender trained upon UserSim performs much more stably than the one trained based upon IRecGAN.

To sum up, the above observations demonstrate that UserSim can effectively mimic user behaviors in real-world recommender systems. Therefore, it has the potential to take the place of real users to train RL-based recommender systems. Note that some **on-policy RL** algorithms such as SARSA [124] cannot be directly trained on historical logs. Thus a simulator is necessary to train these RL algorithms before launching them online.

4.3.4 Effectiveness of Generator

Our proposed generator aims to generate indistinguishable logs (action) based on users' browsing history (state). In other words, it mimics the recommendation policy of the recommender system that generates the historical logs. The aforementioned comparison (UserSim v.s. UserSim-d) proves that the generated logs can enhance feedback predictions. Here, we investigate whether the proposed generator can generate indistinguishable logs. We train several representative recommendation algorithms based on the historical logs, then use them to generate a sequence of recommendations (with the same length of real logs), then compare the sequence similarity of real logs and generated logs. We compare the generator of the proposed generator with the following representative recommender methods:

- **FM**: Factorization Machines [106] combine the advantages of SVMs with factorization models. Compared with matrix factorization, higher-order interactions can be modeled

using the dimensionality parameter.

- **W&D** [24]: This baseline is a wide & deep model for jointly training feed-forward neural networks with embeddings and linear model with feature transformations for generic recommender systems.

- **Autorec** [115]: Autorec learns an autoencoder that encodes each item or user into lower-dimensional space and then decodes to make predictions.

- **GRU4Rec** [55]: GRU4Rec utilizes RNN with GRU units to predict what user will click/order next based on the clicking/ordering histories.

- **RRN** [139]: Recurrent RA Networks predict future behavioral trajectories by endowing both users and movies with a Long Short-Term Memory (LSTM) autoregressive model that captures dynamics.

- **IRGAN** [132]: IRGAN provides a flexible and principled training environment that combines generative and discriminative models for web search, item recommendation, and question answering.

- **SSRM** [52]: propose an MF-based attention model to capture the main intention of a user from her/his historical interactions, and furthermore propose a hybrid session-based recommender to model both a user's long and short-term preferences.

To evaluate the sequence similarity of real and generated logs, we select widely used metrics in NLP community **ROUGE** [37], where ROUGE-N measures the overlap ratio of N-grams between the real and generated sentences. The results are shown in Table 4.2. Compared with baselines, it can observe that the generator of UserSim could generate the

Table 4.2: Generator effectiveness.

Method	**ROUGE-**1	**diff.**	**ROUGE-2**	**diff.**
FM	0.312	-34.5%	0.143	-35.2%
W&D	0.336	-29.3%	0.159	-27.7%
Autorec	0.361	-24.2%	0.165	-24.9%
GRU4Rec	0.380	-20.1%	0.175	-21.2%
RRN	0.415	-13.2%	0.191	-13.6%
IRGAN	0.437	-8.19%	0.202	-8.59%
SSRM	0.452	-5.05%	0.208	-5.88%
UserSim	**0.476**	**-**	**0.221**	**-**

most similar sequence with real-world logs. This result validates that the competition between the generator and discriminator and the proposed supervised components can enhance the generator to capture the complex item distribution in historical logs.

4.3.5 Component Anslysis

To study how the components in the generator and discriminator contribute to the performance, we systematically eliminate the corresponding components of the simulator by defining UserSim's following variants:

- **UserSim-1**: This variant is a simplified version of the simulator who has the same architecture except that the output of the discriminator is a 3-dimensional vector $output = [l_{rp}, l_{rn}, l_f]$, where each logit represents *real-positive*, *real-negative* and *fake* respectively, i.e., it will not distinguish the generated positive and negative items.

- **UserSim-2**: In this variant, we evaluate the contribution of the supervised component L_G^{sup}, so we eliminate the impact of L_G^{sup} by setting $\beta = 0$.

- **UserSim-3**: This variant is to evaluate the adversarial training; hence, we remove L_G^{unsup} and L_D^{unsup} from loss function.

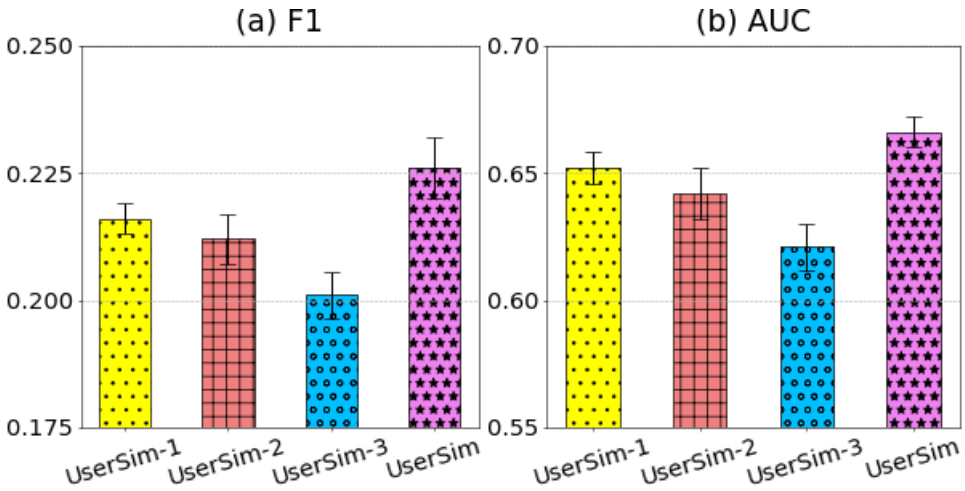

Figure 4.6: The results of component analysis.

The results are shown in Figure 4.6. It can be observed:

- UserSim performs better than UserSim-1, which demonstrates that distinguishing the generated positive and negative items can enhance the performance. This also validates that the generator's output can be considered as augmentations of real-world data, which resolves the data limitation challenge.

- UserSim-2 performs worse than UserSim, which suggests that the supervised component helps the generator to produce more indistinguishable items.

- UserSim-3 first trains a generator, then uses real data and generated data to train the discriminator; while UserSim updates the generator and discriminator iteratively. UserSim outperforms UserSim-3, which indicates that the adversarial training can enhance both the generator (to capture complex data distribution) and the discriminator (to classify real and fake samples).

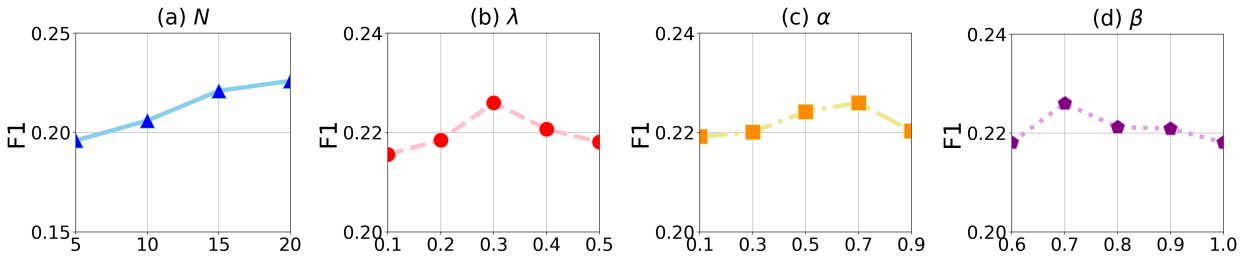

Figure 4.7: The results of parametric analysis.

4.3.6 Parametric Sensitivity Analysis

Our method has several key parameters, i.e., (1) N that controls the length of state; (2) λ that controls the contribution of the second term in Eq. (4.6), which classifies the generated items into positive or negative class; (3) α that adjusts the importance of supervised component in discriminator; and (4) β that calibrates the supervised component in generator. To study the impact of these parameters, we investigate how the proposed framework UserSim works with the changes of one parameter, while fixing other parameters. The results are shown in Figure 4.7. We have following observations:

- Figure 4.7 (a) demonstrates the parameter sensitivity of N. We find that with the increase of N, the performance improves. To be specific, the performance improves significantly first and then becomes relatively stable. This result indicates that introducing longer browsing history can enhance performance.

- Figure 4.7 (b) shows the sensitivity of λ. The performance for the simulator achieves the peak when $\lambda = 0.3$. In other words, the second term in Eq. (4.6) indeed improves the performance of the simulator; however, the performance mainly depends on the first term in Eq. (4.6), which classifies the real items into positive and negative classes.

- Figure 4.7 (c) - (d) illustrate the model performance with respect to α and β. We can

observe that the simulator achieves its optimal performance when $\alpha = 0.7$ and $\beta = 0.7$, while a lower/higher value of both parameters will lead to a lower F1. These observations validate that the supervised components can enhance the performance.

4.4 Related Work

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

The first category related to this chapter is reinforcement learning based recommender systems, which typically consider the recommendation task as a Markov Decision Process (MDP), and model the recommendation procedure as sequential interactions between users and recommender system[159, 155]. A Deep Deterministic Policy Gradient (DDPG) algorithm is introduced to mitigate the large action space issue in practical RL-based recommender systems [32], where an Actor produces the optimal action based on current state, and a Critic outputs the action-value (Q-value) for this state-action pair. Users' positive and negative feedback are jointly considered in one framework to boost recommendations [164]. A page-wise framework is proposed to jointly recommend a page of items and display them within a 2-D page [165, 160]. A multi-agent reinforcement learning based approach (DeepChain) is proposed to jointly optimize multiple recommendation strategies among sequential scenarios [161]. A unified framework is proposed to jointly optimize user experience of recommendations and revenue of advertisements [156, 166]. In news feed scenario, a DQN based framework is proposed to handle the challenges of conventional models [167]. Other applications includes sellers' impression allocation [14], cold-start problem [175], long-term engagement [172] and fairness [42], top-N recommendation [173], attacking black-box recommendations [35] and

spatial recommendation [137].

The second category related to this chapter is behavior simulation. Reinforcement learning and supervised learning algorithms typically learn experts' behavior with the guidance of the rewards, feedback or labels from real-world environment. However, deploying algorithms in real environment cost money and time, which calls for estimation of environment to train the algorithms to learn experts' behavior based on the simulation of the environment, before launching the algorithms online [117, 56, 4, 23]. One of the most effective approaches is Learning from Demonstration (LfD), which estimates implicit reward function from expert's behavior state to action mappings. Successful LfD applications include autonomous helicopter maneuvers [111], self-driving car [8], playing table tennis [15], object manipulation [99] and making coffee [123]. For example, Ross et al. [111] develop a method that autonomously navigates a small helicopter at low altitude in a natural forest environment. Bojarski et al. [8] train a CNN to directly map the raw pixels of a single front-facing camera to the steering commands. Calinon et al. [15] propose a probabilistic method to train robust models of human motion by imitating, e.g., playing table tennis. Sung et al. [123] proposed a manipulation planning approach according to the assumption that many household items share similar operational components. [174] propose a customer simulator, referred to as the World Model, which is designed to simulate the environment and handle the selection bias of logged data.

Chapter 5

Automated Embedding Size Search

Abstract

Practical large-scale recommender systems usually contain thousands of feature fields from users, items, contextual information, and their interactions. Most of them empirically allocate a unified dimension to all feature fields, which is memory inefficient. Thus it is highly desired to assign various embedding dimensions to different feature fields according to their importance and predictability. Due to the large amounts of feature fields and the nuanced relationship between embedding dimensions with feature distributions and neural network architectures, manually allocating embedding dimensions in practical recommender systems can be challenging. To this end, we propose an AutoML-based framework (AutoDim) in this chapter, which can automatically select dimensions for different feature fields in a data-driven fashion. Specifically, we first proposed an end-to-end differentiable framework that can calculate the weights over various dimensions in a soft and continuous manner for feature fields, and an AutoML-based optimization algorithm; then, we derive a hard and discrete embedding component architecture according to the maximal weights and retrain the whole recommender framework. We conduct extensive experiments on benchmark datasets to validate the effectiveness of AutoDim.

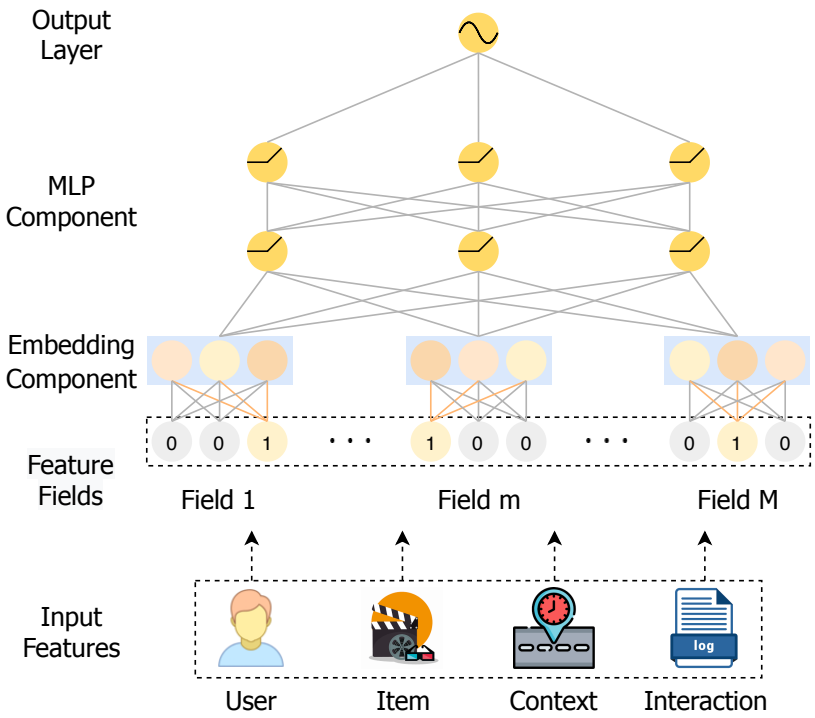

Figure 5.1: The typically DLRS architecture.

5.1 Introduction

Real-world deep learning based recommender systems (DLRSs) typically involve a massive amount of categorical feature fields from users (e.g., occupation and userID), items (e.g., category and itemID), contextual information (e.g., time and location), and their interactions (e.g., user's purchase history of items). DLRSs first map these categorical features into real-valued dense vectors via an *embedding-component* [103, 168], i.e., the embedding-lookup process, which leads to huge amounts of embedding parameters. For instance, the YouTube recommender system consists of 1 million unique videoIDs, and assigns each videoID with a specific 256-dimensional embedding vector; in other words, the videoID feature field alone occupies 256 million parameters [28]. Then, the DLRSs nonlinearly transform the input embeddings from all feature fields and generate the outputs (predictions) via the *MLP-component* (Multi-Layer Perceptron), which usually involves only several fully-connected

layers in practice. Therefore, compared to the MLP-component, the embedding-component dominates the number of parameters in practical DLRSs, which naturally plays a tremendously impactful role in the recommendations.

The majority of existing recommender systems assign a fixed and unified embedding dimension for all feature fields, such as the famous Wide&Deep model [24], which may lead to memory inefficiency. First, the embedding dimension often determines the capacity to encode information. Thus, allocating the same dimension to all feature fields may lose the information of highly predictive features while wasting memory on non-predictive features. Therefore, we should assign a large dimension to the highly informative and predictive features, for instance, the "location" feature in location-based recommender systems [5]. Second, different feature fields have different cardinalities (i.e., the number of unique values). For example, the gender feature has only two (i.e., male and female), while the itemID feature usually involves millions of unique values. Intuitively, we should allocate larger dimensions to the feature fields with more unique feature values to encode their complex relationships with other features, and assign smaller dimensions to feature fields with smaller cardinality to avoid the overfitting problem due to the over-parameterization [44, 63]. According to the above reasons, it is highly desired to assign different embedding dimensions to different feature fields in a memory-efficient manner.

In this chapter, we aim to enable different embedding dimensions for different feature fields for recommendations. We face several tremendous challenges. First, the relationship among embedding dimensions, feature distributions and neural network architectures is highly intricate, which makes it hard to manually assign embedding dimensions to each feature field [44]. Second, real-world recommender systems often involve hundreds and thousands of feature fields. It is difficult, if possible, to artificially select different dimensions for all feature

fields, due to the expensive computation cost from the incredibly huge (N^M, with N the number of candidate dimensions for each feature field, and M the number of feature fields) search space. Our attempt to address these challenges results in an end-to-end differentiable AutoML-based framework (**AutoDim**), which can efficiently allocate embedding dimensions to different feature fields in an automated and data-driven manner. Our experiments on benchmark datasets demonstrate the effectiveness of the proposed framework. We summarize our major contributions as:

- we identify the phenomenon that assigning various embedding dimensions to different feature fields can enhance recommendation performance;
- we propose an end-to-end AutoML-based framework AutoDim, which can automatically select various embedding dimensions to different feature fields; and
- we demonstrate the effectiveness of the proposed framework on real-world benchmark datasets.

5.2 Framework

In this section, we propose an AutoML-based framework, which effectively achieves the automated allocation of varying embedding dimensions to different feature fields. We illustrate the overall framework in Figure 5.2, which contains *dimensionality search stage* and *parameter re-training stage*.

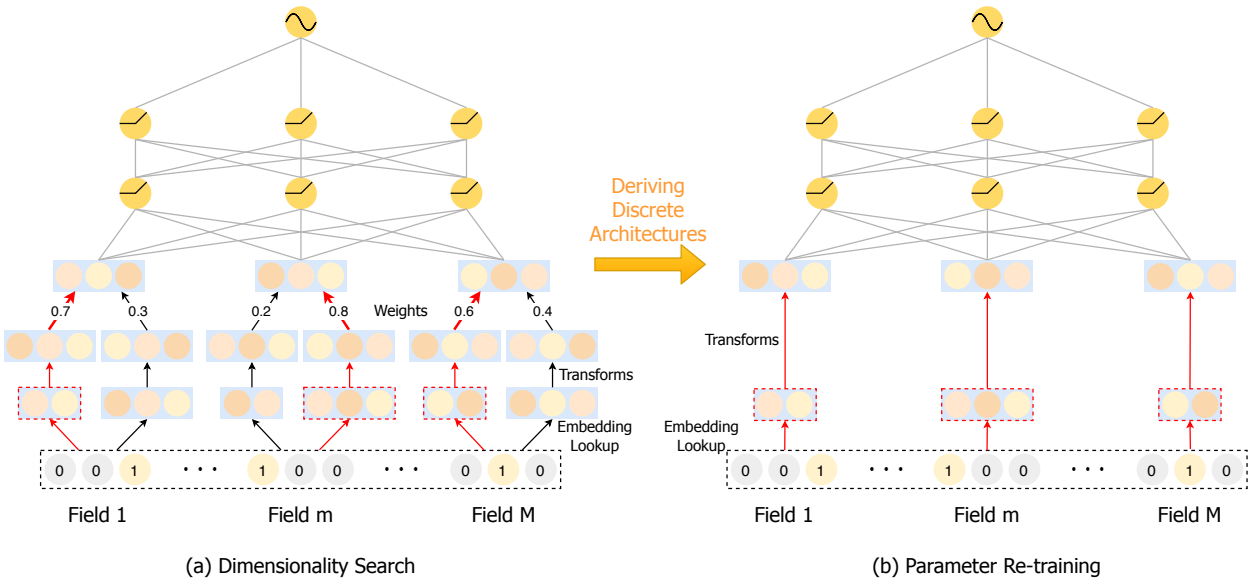

Figure 5.2: Overview of the proposed AutoDim framework.

5.2.1 Dimensionality Search

As the aforementioned challenges in Section 5.1, it is difficult to manually select embedding dimensions via conventional dimension reduction methods. An intuitive solution to tackle this challenge is to assign several embedding spaces with various dimensions to feature fields, and then the DLRS automatically selects the optimal embedding dimension for each feature field.

5.2.1.1 Embedding Lookup

Suppose for each user-item interaction instance, we have M input features $(x_1, \cdots, x_M)$, and each feature x_m belongs to a specific feature field, such as gender and age, etc. For the m^{th} feature field, we assign N candidate embedding spaces $\{\boldsymbol{X}_m^1, \cdots, \boldsymbol{X}_m^N\}$. The dimension of an embedding in each space is $d_1, \cdots, d_N$, where $d_1 < \cdots < d_N$; and the cardinality of these embedding spaces are the number of unique feature values in this feature field. Correspondingly, we define $\{\boldsymbol{x}_m^1, \cdots, \boldsymbol{x}_m^N\}$ as the set of candidate embeddings for a given

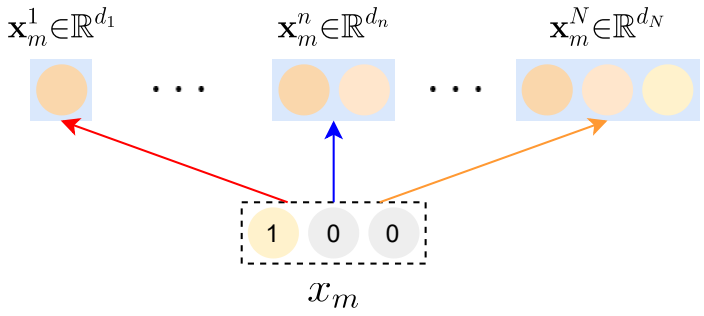

Figure 5.3: Embedding lookup method.

feature x_m from all embedding spaces, as shown in Figure 5.3. Therefore, the total space assigned to the feature x_m is $\sum_{n=1}^{N} d_n$. Note that we assign the same candidate dimensions to all feature fields for simplicity, but it is straightforward to introduce different candidate sets.

5.2.1.2 Unifying Various Dimensions

Since the input dimension of the first MLP layer in existing DLRSs is often fixed, it is difficult for them to handle various candidate dimensions. Thus we need to unify the embeddings $\{\boldsymbol{x}_m^1, \cdots, \boldsymbol{x}_m^N\}$ into same dimension.

Figure 5.4 illustrates the linear transformation method to handle the various embedding dimensions. We introduce N fully-connected layers, which transform embedding vectors $\{\boldsymbol{x}_m^1, \cdots, \boldsymbol{x}_m^N\}$ into the same dimension d_N:

$$\widetilde{\boldsymbol{x}}_m^n \leftarrow \boldsymbol{W}_n^\top \boldsymbol{x}_m^n + \boldsymbol{b}_n \quad \forall n \in [1, N] \tag{5.1}$$

where $\boldsymbol{W}_n \in \mathbb{R}^{d_n \times d_N}$ is weight matrice and $\boldsymbol{b}_n \in \mathbb{R}^{d_N}$ is bias vector. For each field, all candidate embeddings with the same dimension share the same weight matrice and bias vector, which can reduce the amount of model parameters. With the linear transforma-

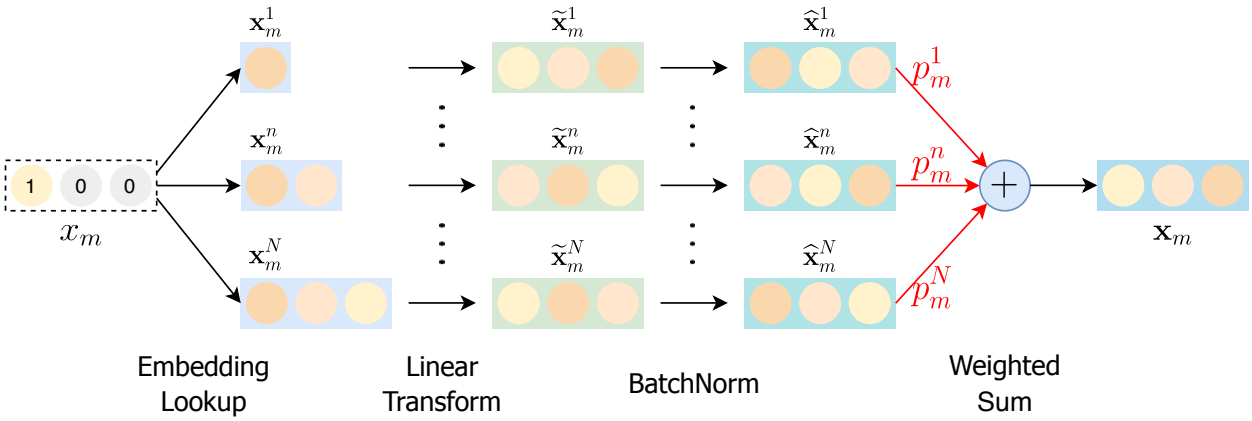

Figure 5.4: Linear transformation method to unify various dimensions.

tions, we map the original embedding vectors $\{\boldsymbol{x}_m^1, \cdots, \boldsymbol{x}_m^N\}$ into the same dimensional space, i.e., $\{\widetilde{\boldsymbol{x}}_m^1, \cdots, \widetilde{\boldsymbol{x}}_m^N\} \in \mathbb{R}^{d_N}$. In practice, we can observe that the magnitude of the transformed embeddings $\{\widetilde{\boldsymbol{x}}_m^1, \cdots, \widetilde{\boldsymbol{x}}_m^N\}$ varies significantly, which makes them become incomparable. To tackle this challenge, we conduct BatchNorm [57] on the transformed embeddings $\{\widetilde{\boldsymbol{x}}_m^1, \cdots, \widetilde{\boldsymbol{x}}_m^N\}$ as:

$$\widehat{\boldsymbol{x}}_m^n \leftarrow \frac{\widetilde{\boldsymbol{x}}_m^n - \mu_{\mathcal{B}}^n}{\sqrt{(\sigma_{\mathcal{B}}^n)^2 + \epsilon}} \quad \forall n \in [1, N] \tag{5.2}$$

where $\mu_{\mathcal{B}}^n$ is the mini-batch mean and $(\sigma_{\mathcal{B}}^n)^2$ is the mini-batch variance for $\forall n \in [1, N]$. ϵ is a small constant added to the mini-batch variance for numerical stability when $(\sigma_{\mathcal{B}}^n)^2$ is very small. After BatchNorm, the linearly transformed embeddings $\{\widetilde{\boldsymbol{x}}_m^1, \cdots, \widetilde{\boldsymbol{x}}_m^N\}$ become to magnitude-comparable embedding vectors $\{\widehat{\boldsymbol{x}}_m^1, \cdots, \widehat{\boldsymbol{x}}_m^N\}$ with the same dimension d_N. Next, we will introduce embedding dimension selection process.

5.2.1.3 Dimension Selection

We aim to select the optimal embedding dimension for each feature field in an automated and data-driven manner. This is a hard (categorical) selection on the candidate embedding spaces, which will make the whole framework not end-to-end differentiable. To tackle this challenge,

in this work, we approximate the hard selection over different dimensions via introducing the Gumbel-softmax operation [58], which simulates the non-differentiable sampling from a categorical distribution by a differentiable sampling from the Gumbel-softmax distribution.

To be specific, suppose weights $\{\alpha_m^1, \cdots, \alpha_m^N\}$ are the class probabilities over different dimensions. Then a hard selection z can be drawn via the the gumbel-max trick [47] as:

$$\begin{aligned} z &= \text{one\_hot}\left(\arg\max_{n\in[1,N]}\left[\log\alpha_m^n + g_n\right]\right) \\ where\ g_n &= -\log\left(-\log\left(u_n\right)\right) \\ u_n &\sim Uniform(0,1) \end{aligned} \tag{5.3}$$

The *gumbel noises* $\{g_i, \cdots, g_N\}$ are i.i.d samples, which perturb $\{\log\alpha_m^n\}$ terms and make the arg max operation that is equivalent to drawing a sample by $\{\alpha_m^1, \cdots, \alpha_m^N\}$ weights. However, this trick is non-differentiable due to the arg max operation. To deal with this problem, we use the softmax function as a continuous, differentiable approximation to arg max operation, i.e., straight-through gumbel-softmax [58]:

$$p_m^n = \frac{\exp\left(\frac{\log(\alpha_m^n)+g_n}{\tau}\right)}{\sum_{i=1}^N \exp\left(\frac{\log\left(\alpha_m^i\right)+g_i}{\tau}\right)} \tag{5.4}$$

where τ is the temperature parameter, which controls the smoothness of the output of gumbel-softmax operation. When τ approaches zero, the output of the gumbel-softmax becomes closer to a one-hot vector. Then p_m^n is the probability of selecting the n^{th} candidate embedding dimension for the feature x_m, and its embedding $\boldsymbol{x}_m$ can be formulated as the

weighted sum of $\{\widehat{\boldsymbol{x}}_m^1, \cdots, \widehat{\boldsymbol{x}}_m^N\}$:

$$\boldsymbol{x}_m = \sum_{n=1}^{N} p_m^n \cdot \widehat{\boldsymbol{x}}_m^n \quad \forall m \in [1, M] \tag{5.5}$$

We illustrate the weighted sum operations in Figure 5.4. With gumbel-softmax operation, the hard-like dimensionality search process is end-to-end differentiable. The discrete embedding dimension selection conducted based on the weights $\{\alpha_m^n\}$ will be detailed in the following subsections.

Then, we concatenate the embeddings $\boldsymbol{h}_0 = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_M]$ and feed $\boldsymbol{h}_0$ input into L multilayer perceptron layers:

$$\boldsymbol{h}_l = \sigma\left(\boldsymbol{W}_l^\top \boldsymbol{h}_{l-1} + \boldsymbol{b}_l\right) \quad \forall l \in [1, L] \tag{5.6}$$

where $\boldsymbol{W}_l$ and $\boldsymbol{b}_l$ are the weight matrix and the bias vector for the l^{th} MLP layer. $\sigma(\cdot)$ is the activation function such as *ReLU* and *Tanh*. Finally, the output layer that is subsequent to the last MLP layer, produces the prediction of the current user-item interaction instance as:

$$\hat{y} = \sigma\left(\boldsymbol{W}_o^\top \boldsymbol{h}_L + \boldsymbol{b}_o\right) \tag{5.7}$$

where $\boldsymbol{W}_o$ and $\boldsymbol{b}_o$ are the weight matrix and bias vector for the output layer. Activation function $\sigma(\cdot)$ is selected based on different recommendation tasks, such as Sigmoid for regression [24], and Softmax for multi-class classification [128]. Correspondingly, the objective function $\mathcal{L}(\hat{y}, y)$ between prediction $\hat{y}$ and ground truth label y also varies. In this work, we

leverage negative log-likelihood function:

$$\mathcal{L}(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) \tag{5.8}$$

where y is the ground truth (1 for like or click, 0 for dislike or non-click). By minimizing the objective function $\mathcal{L}(\hat{y}, y)$, the dimensionality search framework updates the parameters of all embeddings, hidden layers, and weights $\{\alpha_m^n\}$ through back-propagation. The high-level idea of the dimensionality search is illustrated in Figure 5.2 (a), where we omit some details of embedding-lookup, transformations and gumbel-softmax for the sake of simplicity.

5.2.2 Optimization

In this subsection, we will detail the optimization method of the proposed AutoDim framework. In AutoDim, we formulate the selection over different embedding dimensions as an architectural optimization problem and make it end-to-end differentiable by leveraging the Gumbel-softmax technique. The parameters to be optimized in AutoDim are two-fold, i.e., (i) $\boldsymbol{W}$: the parameters of the DLRS, including the embedding-component and the MLP-component; (ii) $\boldsymbol{\alpha}$: the architectural weights $\{\alpha_m^n\}$ on different embedding spaces ($\{p_m^n\}$ are calculated based on $\{\alpha_m^n\}$ as in Eq. (5.4)). DLRS parameters $\boldsymbol{W}$ and architectural weights $\boldsymbol{\alpha}$ can not be optimized simultaneously on the training dataset as with the conventional supervised attention mechanism since theie optimization are highly dependent on each other. In other words, optimization on the training dataset simultaneously may result in the model overfitting on the examples from the training dataset.

Inspired by the differentiable architecture search (DARTS) techniques [80], $\boldsymbol{W}$ and $\boldsymbol{\alpha}$ are alternately optimized through gradient descent. Specifically, we alternately update $\boldsymbol{W}$ by

Algorithm 5.1: DARTS based Optimization for AutoDim.

Input: the features $(x_1, \cdots, x_M)$ of user-item interactions and the corresponding ground-truth labels y
Output: the well-learned DLRS parameters $\boldsymbol{W}^*$; the well-learned weights on various embedding spaces $\boldsymbol{\alpha}^*$

```
while not converged do
    Sample a mini-batch of user-item interactions from validation data
    Update α by descending ∇_α L_val(W*(α), α)
    Collect a mini-batch of training data
    Generate predictions ŷ via DLRS with current W and architectural weights α
    Update W by descending ∇_W L_train(W, α)
end while
```

optimizing the loss $\mathcal{L}_{train}$ on the training data and update $\boldsymbol{\alpha}$ by optimizing the loss $\mathcal{L}_{val}$ on the validation data:

$$\begin{aligned} &\min_{\boldsymbol{\alpha}} \mathcal{L}_{val}\big(\boldsymbol{W}^*(\boldsymbol{\alpha}), \boldsymbol{\alpha}\big) \\ &s.t.\ \boldsymbol{W}^*(\boldsymbol{\alpha}) = \arg\min_{\boldsymbol{W}} \mathcal{L}_{train}(\boldsymbol{W}, \boldsymbol{\alpha}^*) \end{aligned} \tag{5.9}$$

this optimization forms a bilevel optimization problem [100], where architectural weights $\boldsymbol{\alpha}$ and DLRS parameters $\boldsymbol{W}$ are identified as the upper-level variable and lower-level variable. Since the inner optimization of $\boldsymbol{W}$ is computationally expensive, directly optimizing $\boldsymbol{\alpha}$ via Eq. (5.9) is intractable. To address this challenge, we take advantage of the approximation scheme of DARTS:

$$\arg\min_{\boldsymbol{W}} \mathcal{L}_{train}(\boldsymbol{W}, \boldsymbol{\alpha}^*) \approx \boldsymbol{W} - \xi \nabla_{\boldsymbol{W}} \mathcal{L}_{train}(\boldsymbol{W}, \boldsymbol{\alpha}) \tag{5.10}$$

where ξ is the learning rate. In the approximation scheme, when updating $\boldsymbol{\alpha}$ via Eq. (5.10), we estimate $\boldsymbol{W}^*(\boldsymbol{\alpha})$ by descending the gradient $\nabla_{\boldsymbol{W}} \mathcal{L}_{train}(\boldsymbol{W}, \boldsymbol{\alpha})$ for only one step, rather than to optimize $\boldsymbol{W}(\boldsymbol{\alpha})$ thoroughly to obtain $\boldsymbol{W}^*(\boldsymbol{\alpha}) = \arg\min_{\boldsymbol{W}} \mathcal{L}_{train}(\boldsymbol{W}, \boldsymbol{\alpha}^*)$.

The DARTS based optimization algorithm for AutoDim is detailed in Algorithm 5.1. Specifically, in each iteration, we first sample a batch of user-item interaction data from

the validation set (line 2); next, we update the architectural weights $\boldsymbol{\alpha}$ upon it (line 3); afterward, the DLRS make the predictions $\hat{y}$ on the batch of training data with current DLRS parameters $\boldsymbol{W}$ and architectural weights $\boldsymbol{\alpha}$ (line 4-5); eventually, we update the DLRS parameters $\boldsymbol{W}$ by descending $\nabla_{\boldsymbol{W}} \mathcal{L}_{train}(\boldsymbol{W}, \boldsymbol{\alpha})$ (line 6).

5.2.3 Parameter Re-Training

Since the suboptimal embedding dimensions in the dimensionality search stage also influence the model training, a retraining stage is desired to train the model with only optimal dimensions, eliminating these suboptimal influences. This subsection will introduce how to select the optimal embedding dimension for each feature field and the details of retraining the recommender system with the selected embedding dimensions.

5.2.3.1 Deriving Discrete Dimensions

During re-training, the gumbel-softmax operation is no longer used, which means that the optimal embedding space (dimension) are selected for each feature field as the one corresponding to the largest weight, based on the well-learned $\boldsymbol{\alpha}$. It is formally defined as:

$$\boldsymbol{X}_m = \boldsymbol{X}_m^k, \quad where \;\; k = \arg\max_{n \in [1,N]} \alpha_m^n \quad \forall m \in [1, M] \tag{5.11}$$

Figure 5.2 (a) illustrates the architecture of AutoDim framework with a toy example about the optimal dimension selections based on two candidate dimensions, where the largest weights corresponding to the 1^{st}, m^{th} and M^{th} feature fields are 0.7, 0.8 and 0.6, then the embedding space $\boldsymbol{X}_1^1$, $\boldsymbol{X}_m^2$ and $\boldsymbol{X}_M^1$ are selected for these feature fields. The dimension of an embedding vector in these embedding spaces is d_1, d_2 and d_1, respectively.

Algorithm 5.2: The Optimization of DLRS Re-training Process.

Input: the features $(x_1, \cdots, x_M)$ of user-item interactions and the corresponding ground-truth labels y

Output: the well-learned DLRS parameters $\boldsymbol{W}^*$

1: **while** not converged **do**
2: Sample a mini-batch of training data
3: Generate predictions $\hat{y}$ via DLRS with current $\boldsymbol{W}$
4: Update $\boldsymbol{W}$ by descending $\nabla_{\boldsymbol{W}} \mathcal{L}_{train}(\boldsymbol{W})$
5: **end while**

5.2.3.2 Model Re-training

As shown in Figure 5.2 (b), given the selected embedding spaces, we can obtain unique embedding vectors $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_M)$ for features $(x_1, \cdots, x_M)$. Then we concatenate these embeddings and feeds them into hidden layers. Next, the prediction $\hat{y}$ is generated by the output layer. Finally, all the parameters of the DLRS, including embeddings and MLPs, will be updated via minimizing the supervised loss function $\mathcal{L}(\hat{y}, y)$ through back-propagation. The model retraining algorithm is detailed in Algorithm 5.2. The retraining process is based on the same training data as Algorithm 5.1.

Note that the majority of existing deep recommender algorithms (such as FM [106], DeepFM [51], xDeepFM [74]) capture the interactions between feature fields via interaction operations, such as inner product. These interaction operations require the embedding vectors from all fields to have the same dimensions. Therefore, the embeddings selected in Section 5.2.3.1 are still mapped into the same dimension as in Section 5.2.1.2. In the retraining stage, the BatchNorm operation is no longer in use, since there are no competitions between candidate embeddings in each field.

5.3 Experiments

In this section, we first introduce experimental settings. Then we conduct extensive experiments to evaluate the effectiveness of the proposed AutoDim framework.

5.3.1 Dataset

We evaluate our model on benchmark Criteo dataset[1]: This is a benchmark industry dataset to evaluate ad click-through rate prediction models. It consists of 45 million users' click records on displayed ads over one month. For each data example, it contains 13 numerical feature fields and 26 categorical feature fields. We normalize numerical features by transforming a value $v \rightarrow \lfloor \log(v)^2 \rfloor$ if $v > 2$ as proposed by the Criteo Competition winner[2], and then convert it into categorical features through bucketing. All $M = 39$ feature fields are anonymous. We use 90% user-item interactions as the training/validation set (8:1), and the rest 10% as the test set.

5.3.2 Implement Details

Next, we detail the AutoDim architectures. For the DLRS, (i) embedding component: (ii) MLP component: we have two hidden layers with the size $|h_0| \times 128$ and 128×128, where $|h_0|$ is the input size of first hidden layer, $|h_0| = 32 \times M$ with $M = 39$ the number of feature fields for Criteo dataset, and we use batch normalization, dropout ($rate = 0.2$) and ReLU activation for both hidden layers. The output layer is 128×1 with Sigmoid activation.

For architectural weights $\boldsymbol{\alpha}$: $\alpha_m^1, \cdots, \alpha_m^N$ of the m^{th} feature field are produced by a Softmax activation upon a trainable vector of length N. We use an annealing temperature

[1] https://www.kaggle.com/c/criteo-display-ad-challenge/

[2] https://www.csie.ntu.edu.tw/ r01922136/kaggle-2014-criteo.pdf

$\tau = \max(0.01, 1 - 0.00005 \cdot t)$ for Gumbel-softmax, where t is the training step.

The learning rate for updating DLRS and weights are 0.001 and 0.001, and the batch-size is set as 2000. Our model can be applied to **any deep recommender systems with embedding layers**. In this chapter, we show the performances of applying AutoDim on the well-known FM [106], W&D [24] and DeepFM [51].

5.3.3 Evaluation Metrics

The performance is evaluated by AUC, Logloss and Params, where a higher AUC or a lower Logloss indicates a better recommendation performance. A lower Params means fewer embedding parameters. A slightly higher AUC or lower Logloss at 0.001-level is regarded as significant for the CTR prediction task [24, 51]. For an embedding dimension search model, the "Params" metric is the optimal number of embedding parameters selected by this model for the recommender system. We omit the number of MLP parameters, which only occupy a small part of the total model parameters, e.g., $\sim 0.5\%$ in W&D and DeepFM on Criteo dataset. FM model has no MLP component.

5.3.4 Overall Performance

We compare the proposed framework with following embedding dimension search methods:

- **FDE** (Full Dimension Embedding): In this baseline, we assign the maximal candidate dimension to all feature fields, i.e., 32. For each feature field, the embedding dimension is set as the candidate set's maximal size
- **MDE** (Mixed Dimension Embedding) [44]: This is a heuristic method that assigns highly-frequent feature values with larger embedding dimensions, vice versa. We enumerate its 16

groups of suggested hyperparameters settings and report the best one.

- **DPQ** (Differentiable Product Quantization) [22]: This baseline introduces differentiable quantization techniques from *network compression* community to compact embeddings.
- **NIS** (Neural Input Search) [62]: This baseline applies reinforcement learning to learn to allocate larger embedding sizes to active feature values, and smaller sizes to inactive ones.
- **MGQE** (Multi-granular quantized embeddings) [63]: This baseline is based on DPQ, and further cuts down the embeddings space by using fewer centroids for non-frequent feature values.
- **AEmb** (Automated Embedding Dimensionality Search) [158]: This baseline is based on DARTS [80], and assigns embedding dimensions according to the frequencies of feature values.
- **RaS** (Random Search): Random search is strong baseline in neural network search [80]. We apply the same candidate embedding dimensions, randomly allocate dimensions to feature fields in each experiment time, and report the best performance.
- **AD-s**: This baseline shares the same architecture with AutoDim, while we update the DLRS parameters and architectural weights simultaneously on the same training batch in an end-to-end backpropagation fashion.

The overall results are shown in Table 5.1. We can observe:

- FDE achieves the worst recommendation performance and largest Params, where FDE is assigned the maximal embedding dimension 32 to all feature fields. This result demonstrates that allocating the same dimension to all feature fields is not only memory inefficient, but introduces numerous noises into the model.

Table 5.1: Performance comparison of different embedding search methods.

Model	Metrics	Search Methods								
		FDE	MDE	DPQ	NIS	MGQE	AEmb	RaS	AD-s	AutoDim
FM	AUC	0.8020	0.8027	0.8035	0.8042	0.8046	0.8049	0.8056	0.8063	**0.8078***
	Logloss	0.4487	0.4481	0.4472	0.4467	0.4462	0.4460	0.4457	0.4452	**0.4438***
	Params	34.778	15.520	20.078	13.636	12.564	13.399	16.236	31.039	**11.632***
W&D	AUC	0.8045	0.8051	0.8058	0.8067	0.8070	0.8072	0.8076	0.8081	**0.8098***
	Logloss	0.4468	0.4464	0.4457	0.4452	0.4446	0.4445	0.4443	0.4439	**0.4419***
	Params	34.778	18.562	22.628	14.728	15.741	15.987	18.233	30.330	**12.455***
DeepFM	AUC	0.8056	0.8060	0.8067	0.8076	0.8080	0.8082	0.8085	0.8089	**0.8101***
	Logloss	0.4457	0.4456	0.4449	0.4442	0.4439	0.4438	0.4436	0.4432	**0.4416***
	Params	34.778	17.272	25.737	12.955	13.059	13.437	17.816	31.770	**11.457***

"*" indicates the statistically significant improvements (i.e., two-sided t-test with $p < 0.05$) over the best baseline. (Params/Million)

- RaS, AD-s, AutoDim performs better than MDE, DPQ, NIS, MGQE, AEmb. The major differences between these two groups of methods are: (i) the first group aims to assign different embedding dimensions to different feature fields, while embeddings in the same feature field share the same dimension; (ii) the second group attempts to assign different embedding sizes to different feature values within the same feature fields, which are based on the frequencies of feature values. The second group of methods surfer from several challenges: (ii-a) there are numerous unique values in each feature field, e.g., 2.7×10^4 values for each feature field on average in the Criteo dataset. This leads to a huge search space (even after bucketing) in each feature field, which makes it difficult to find the optimal solution, while the search space for each feature field is $N = 5$ in AutoDim; (ii-b) allocating dimensions solely based on feature frequencies (i.e., how many times a feature value appears in the training set) may lose other important characteristics of the feature; and (ii-c) the feature values frequencies are usually dynamic and not pre-known in real-time recommender systems, e.g., the cold-start users/items.

- AutoDim outperforms RaS and AD-s, where AutoDim updates the architectural weights

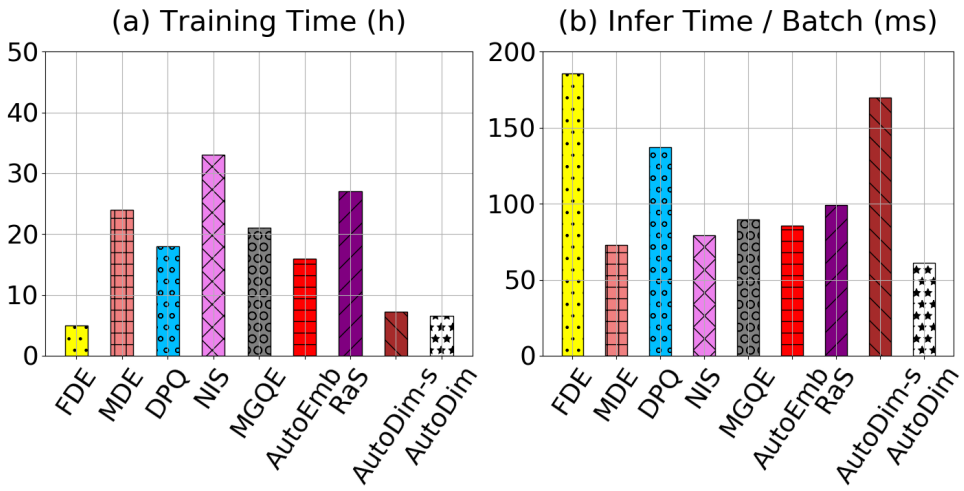

Figure 5.5: Efficiency analysis of DeepFM on Criteo dataset.

$\boldsymbol{\alpha}$ on the validation batch, which can enhance the generalization; AD-s updates the $\boldsymbol{\alpha}$ with DLRS on the same training batch simultaneously, which may lead to overfitting; RaS randomly search the dimensions, which has a large search space. AD-s has much larger Params than AutoDim, which indicates that larger dimensions are more efficient in minimizing training loss.

To sum up, compared with the representative baselines, AutoDim achieves significantly better recommendation performance, and saves 70% $\sim$ 80% embedding parameters. These results prove the effectiveness of the AutoDim framework.

5.3.5 Efficiency Analysis

In addition to model effectiveness, training and inference efficiency are also essential metrics for deploying a recommendation model into commercial recommender systems. This section investigates the efficiency of applying search methods to DeepFM on the Criteo dataset (on one Tesla K80 GPU). We illustrate the results in Figure 5.5:

- For the training time in Figure 5.5 (a), we can observe that AutoDim and AD-s have fast

training speed. As discussed in Section 5.3.4, the reason is that they have a smaller search space than other baselines. FDE's training is fastest since we directly set its embedding dimension as 32, i.e., no searching stage, while its recommendation performance is worst among all methods in Section 5.3.4.

- For the inference time, which is more crucial when deploying a model in commercial recommender systems, AutoDim achieves the least inference time, as shown in Figure 5.5 (b). This is because the final recommendation model selected by AutoDim has the least embedding parameters, i.e., the Params metric. To summarize, AutoDim can efficiently achieve better performance, making it easier to be launched in real-world recommender systems.

5.3.6 Parameter Analysis

In this section, we investigate how essential hyper-parameters influence model performance. Besides common hyper-parameters of deep recommender systems such as the number of hidden layers (we omit them due to limited space), our model has one particular hyper-parameter, i.e., the frequency to update architectural weights $\boldsymbol{\alpha}$, referred to as f. In Algorithm 5.1, we alternately update DLRS's parameters on the training data and update $\boldsymbol{\alpha}$ on the validation data. In practice, we find that updating $\boldsymbol{\alpha}$ can be less frequently than updating DLRS's parameters, which apparently reduces lots of computations, and also enhances the performance.

To study the impact of f, we investigate how DeepFM with AutoDim performs on Criteo dataset with the changes of f, while fixing other parameters. Figure 5.6 shows the parameter sensitivity results, where in x-axis, $f = i$ means updating $\boldsymbol{\alpha}$ once, then updating DLRS's

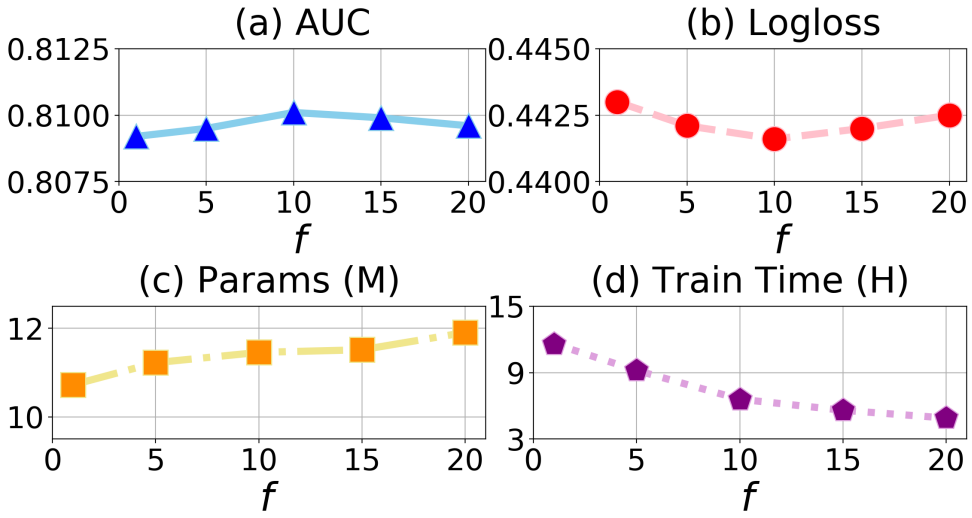

Figure 5.6: Parameter analysis on Movielens-1m dataset.

parameters i times. We can observe that:

- AutoDim achieves the optimal AUC/Logloss when $f = 10$. In other words, updating $\boldsymbol{\alpha}$ too frequently/infrequently results in suboptimal performance. Figure 5.6 (d) shows that setting $f = 10$ can reduce $\sim 50\%$ training time compared with setting $f = 1$.

- Figure 5.6 (c) shows that lower f leads to lower Params, vice versa. The reason is that AutoDim updates $\boldsymbol{\alpha}$ by minimizing validation loss, which improves the generalizability of model [100, 80]. When updating $\boldsymbol{\alpha}$ frequently (e.g., $f = 1$), AutoDim tends to select a smaller embedding size that has better generalization, while may has an under-fitting problem; while when updating $\boldsymbol{\alpha}$ infrequently (e.g., $f = 20$), AutoDim prefers larger embedding sizes that perform better on the training set, but may lead to an over-fitting problem. $f = 10$ is a good trade-off between model performance on training and validation sets.

Table 5.2: Embedding dimensions for Movielens-1m.

feature field	W&D (one field)		AutoDim
	AUC	Logloss	Dimension
movieId	0.7321	0.5947	8
year	0.5763	0.6705	2
genres	0.6312	0.6536	4
userId	0.6857	0.6272	8
gender	0.5079	0.6812	2
age	0.5245	0.6805	2
occupation	0.5264	0.6805	2
zip	0.6524	0.6443	4

5.3.7 Case Study

In this section, we investigate whether AutoDim can assign larger embedding dimensions to more important features. Since feature fields are anonymous in Criteo, we apply W&D with AutoDim on MovieLens-1m dataset[3]. There are $M = 8$ categorical feature fields: movieId, year, genres, userId, gender, age, occupation, zip. Since MovieLens-1m is much smaller than Criteo, we set the candidate embedding dimensions as $\{2, 4, 8, 16\}$.

To measure the contribution of a feature field to the final prediction, we build a W&D model with only this field, train this model and evaluate it on the test set. A higher AUC and a lower Logloss means this feature field is more predictive for the final prediction. Then, we build a comprehensive W&D model incorporating all feature fields, and apply AutoDim to select the dimensions. The results are shown in Table 5.2. It can be observed that:

- No feature fields are assigned 16-dimensional embedding space, which means candidate embedding dimensions $\{2, 4, 8, 16\}$ are sufficient to cover all possible choices.
- Compared to the AUC/Logloss of W&D with each feature field, we can find that AutoDim assigns larger embedding dimensions to important (highly predictive) feature fields, such

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

as movieId and userId, vice versa.

- We build a full dimension embedding (FDE) version of W&D, where all feature fields are assigned as the maximal dimension 16. Its performances are AUC=0.8077, Logloss=0.5383, while the performances of W&D with AutoDim are AUC=0.8113, Logloss=0.5242, and it saves 57% embedding parameters.

In short, above observations validates that AutoDim can assign larger embedding dimensions to more predictive feature fields.

5.4 Related Work

In this section, we will discuss the related works. We summarize the works related to our research from two perspectives: deep recommender systems and AutoML for neural architecture search.

Deep recommender systems have drawn increasing attention from both academia and the industry thanks to their great advantages over traditional methods [154]. Various types of deep learning approaches in recommendation are developed. Sedhain et al. [115] present an AutoEncoder based model named AutoRec. Hidasi et al. [55] introduce an RNN based recommender system named GRU4Rec. Cheng et al. [24] introduce a Wide&Deep framework for both regression and classification tasks. Guo et al. [51] propose the DeepFM model. It combines the factorization machine (FM) and MLP. Wang et al. [133] utilizes CNN to extract visual features to help POI (Point-of-Interest) recommendations. Wang et al. [132] propose a generative adversarial network (IRGAN) based information retrieval model. Zhao et al. [35, 42, 155, 175] propose a series of deep reinforcement learning based recommendation models.

The research of AutoML for neural architecture search can be traced back to NAS [170], which first utilizes an RNN based controller to design neural networks and proposes a reinforcement learning algorithm to optimize the framework. After that, many endeavors are conducted to reduce the high training cost of NAS. Pham et al. [100] propose ENAS, where the controller learns to search a subgraph from a large computational graph to form an optimal neural network architecture. Brock et al. [11] introduce a framework named SMASH, in which a hyper-network is developed to generate weights for sampled networks. DARTS [80] and SNAS [144] formulate the problem of network architecture search in a differentiable manner and solve it using gradient descent. Luo et al. [87] investigate representing network architectures as embeddings. Some works raise another way of thinking, which is to limit the search space. Zoph et al. [171] propose a transfer learning framework called NASNet, which trains convolution cells on smaller datasets and applies them on larger datasets. Tan et al. [127] introduce MNAS. They propose to search hierarchical convolution cell blocks independently, so that a deep network can be built based on them.

Chapter 6

Automated Loss Function Search

Abstract

Designing an effective loss function plays a crucial role in training deep recommender systems. Most existing works often leverage a predefined and fixed loss function that could lead to suboptimal recommendation quality and training efficiency. Some recent efforts rely on exhaustively or manually searched weights to fuse a group of candidate loss functions, which is exceptionally costly in computation and time. They also neglect the various convergence behaviors of different data examples. In this work, we propose an AutoLoss framework that can automatically and adaptively search for the appropriate loss function from a set of candidates. To be specific, we develop a novel controller network, which can dynamically adjust the loss probabilities in a differentiable manner. Unlike existing algorithms, the proposed controller can adaptively generate the loss probabilities for different data examples according to their varied convergence behaviors. Such design improves the model's generalizability and transferability between deep recommender systems and datasets. We evaluate the proposed framework on two benchmark datasets. The results show that AutoLoss outperforms representative baselines. Further experiments have been conducted to deepen our understandings of AutoLoss, including its transferability, components and training efficiency.

6.1 Introduction

In the era of information explosion, recommender systems play a pivotal role in alleviating information overload, which vastly enhance user experiences in many commercial applications, such as generating playlists in video and music services [156, 166], recommending products in online stores [161, 175, 35, 160, 164], and suggesting locations for geo-social events [85, 163, 49]. With the recent growth of deep learning techniques, there have been increasing interests in developing deep recommender systems (DRS) [97, 142]. DRS has improved the recommendation quality since they can effectively learn feature representations and capture the nonlinear relationships between users and items via deep architectures [154]. Aside from developing sophisticated neural network architectures, well-designed loss functions have also been demonstrated to be effective in improving the performance in different recommendation tasks, such as item rating prediction (regression) [105], click-through rate prediction (binary classification) [51, 42], user behavior prediction (multi-class classification) [162], and item retrieval (clustering) [40].

To optimize DRS frameworks, most existing works are based on a predefined and fixed loss function, such as *mean-squared-error* (MSE) or *mean-absolute-error* (MAE) loss for regression tasks. Then DRS frameworks are optimized in a back-propagation manner, which computes gradients effectively and efficiently to minimize the given loss on the training dataset. During this process, the key step is to calculate gradients of network parameters for minimizing loss functions. However, it is often unclear whether the gradients generated from a given loss function are optimal. For example, in regression tasks, the MSE loss can ensure that the trained model has no outlier predictions with huge errors, while MAE performs better if we only want a well-rounded model that performs well on the majority [33, 17].

Therefore, solely utilizing a predefined and fixed loss function for all *data examples*, i.e., *user-item interactions*, cannot guarantee the optimal gradients throughout, especially when the interactions have varied convergence behaviors in the non-stationary environment of online recommendation platforms. In addition, there is often a gap between model training and evaluation performance in real-world recommender systems. For instance, we usually train a predictive model by minimizing *cross-entropy loss* in online advertising, and evaluate the model performance by *click-through rate* (CTR). Consequently, it naturally raises a question - can we incorporate more loss functions in the training phase to enhance the model performance?

Efforts have been made to develop strategies to fuse multiple loss functions, which can take advantage of multiple loss functions in a weighted sum fashion. For example, Panoptic FPN [65] leverages a grid search to find better loss weights; and UPSNet [145] carefully investigates the weighting scheme of loss functions. However, these works rely on exhaustively or manually search for loss weights from a large candidate space, which would be an extremely costly execution in both computing power and time. Also, they aim to learn a set of unified and static weights over the loss functions, which entirely overlook the different convergence behaviors of data examples. Finally, retraining loss weights is always desired when switching among different DRS frameworks or recommendation datasets, which reduces their generalizability and transferability.

In order to obtain more accurate gradients to improve the recommendation performance and the training efficiency, we propose an automated loss function search framework, **AutoLoss**, which can dynamically and adaptively select appropriate loss functions for training DRS frameworks. Different from existing searching models with predefined and fixed loss functions, or the loss weights exhaustively or manually searched, the optimal loss function in

AutoLoss is automatically selected for each data example in a differentiable manner. The experiments on two datasets demonstrate the effectiveness of the proposed framework. We summarize our major contributions as follows:

- We propose an end-to-end framework, AutoLoss, which can automatically select the proper loss functions for training DRS frameworks with better recommendation performance and training efficiency;
- A novel controller network is developed to adaptively adjust the probabilities over multiple loss functions according to different data examples' dynamic convergence behaviors during training, which enhances the model generalizability between different DRS frameworks and datasets;
- We empirically demonstrate the effectiveness of the proposed framework on real-world benchmark datasets. Extensive studies verify the importance of model components and the transferability of AutoLoss.

6.2 The Proposed Framework

In this section, we will present an end-to-end framework, AutoLoss, which effectively tackles the aforementioned challenges in Section 6.1 via automatically and adaptively searching the optimal loss function from several candidates according to data examples' convergence behaviors. We will first provide an overview of the framework; next detail the architectures of the main DRS network; then introduce the loss function search method with a novel controller network; and finally provide an AutoML-based optimization algorithm.

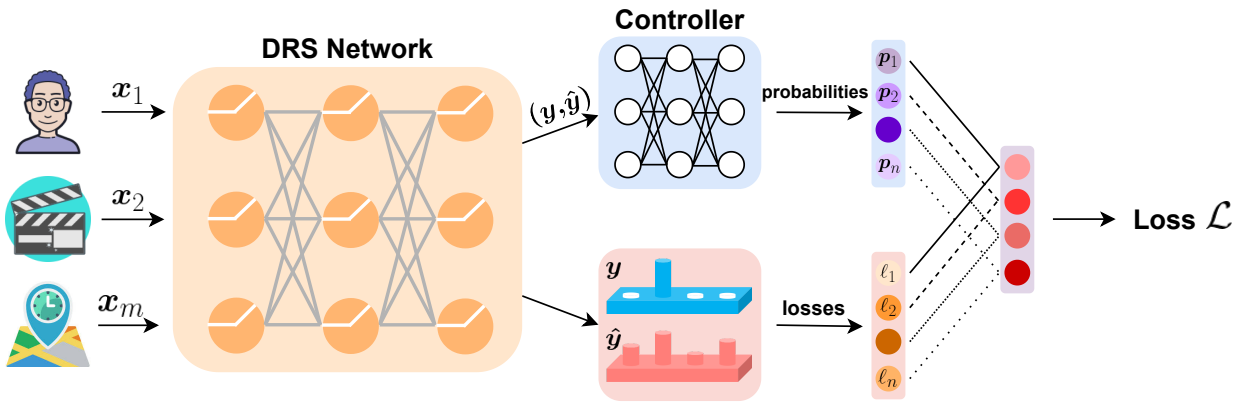

Figure 6.1: Overview of the AutoLoss framework.

6.2.1 An Overview

In this subsection, we will give an overview of the AutoLoss framework. AutoLoss aims to automatically select appropriate loss functions from a set of candidates for different data examples (i.e., user-item interactions). We demonstrate the framework in Figure 6.1. With a DRS network, a controller network and a set of predefined candidate loss functions, the learning process of AutoLoss mainly consists of two major steps.

The forward-propagation step. Given a mini-batch of data examples, the main DRS network first generates predictions $\hat{\boldsymbol{y}}$ based on the input features $\boldsymbol{x}$. Then, we can calculate the losses $\{\ell_i\}$ for each candidate loss function according to the ground truth labels $\boldsymbol{y}$ and predictions $\hat{\boldsymbol{y}}$. Meanwhile, the controller network takes $(\boldsymbol{y}, \hat{\boldsymbol{y}})$ and outputs the probabilities $\boldsymbol{p}$ over loss functions for each data example. Finally, the overall loss $\mathcal{L}$ can be calculated according to the losses from $\{\ell_i\}$ and the probabilities $\boldsymbol{p}$.

The backward-propagation step. We first fix the parameters of the controller network and update the main DRS network parameters upon the training data examples. Then, we fix the DRS parameters and optimize the controller network parameters based on a mini-batch of validation data examples. This alternative updating approach enhances the

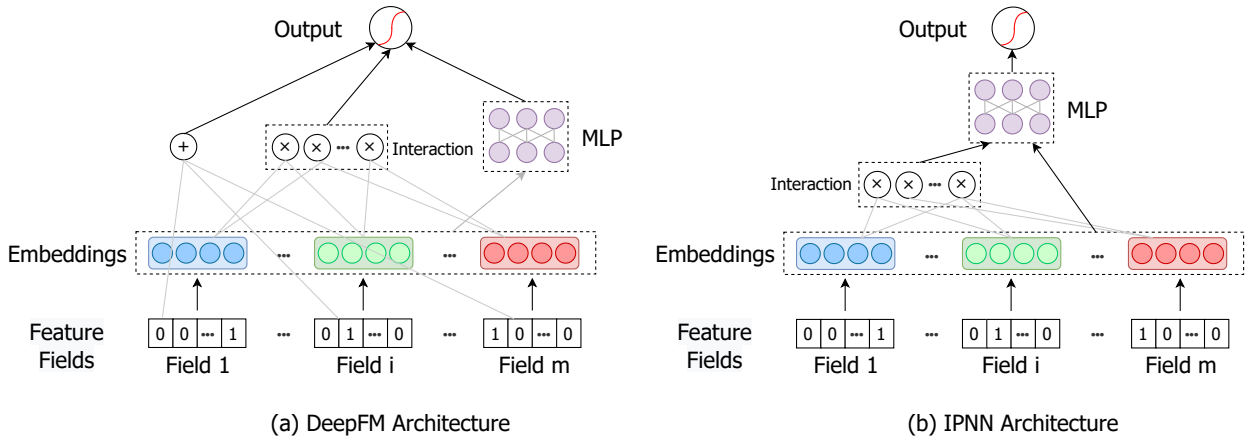

Figure 6.2: Architectures of DeepFM and IPNN.

generalizability, and prevents AutoLoss from selecting probabilities that overfit the training data examples [100, 80]. Next, we will introduce the details of AutoLoss.

6.2.2 Deep Recommender System Network

AutoLoss is quite general for most existing deep recommender system frameworks [106, 51, 74, 103]. As visualized in Figure 6.2, they typically have four components: embedding layer, interaction layer, MLP layer and output layer. We now briefly introduce these components.

6.2.2.1 Embedding Layer

The raw input features of users and items are usually categorical or numeric, and in the form of multiple fields. Most DRS works first transform the input features into binary vectors, and then embed them into continuous vectors using a field-wise embedding. In this way, a user-item interaction data example $\boldsymbol{x} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_m]$ can be represented as the concatenation of binary vectors from all feature fields:

$$\underbrace{[0,1,0,0,\ldots,0]}_{\boldsymbol{x}_1:\text{userid}}\underbrace{[\quad 1,0 \quad]}_{\boldsymbol{x}_2:\text{gender}}\underbrace{[0,1,0,0]}_{\boldsymbol{x}_3:\text{age}}\overbrace{\cdots\cdots}^{\text{other fields}}\underbrace{[0,1,0,1,\ldots,0]}_{\boldsymbol{x}_m:\text{itemid}}$$

where m is the number of feature fields and $\boldsymbol{x}_i$ is the binary vector of the i^{th} field. The categorical data are transformed into binary vectors via one-hot encoding, e.g., $[0, 1]$ for $gender = Female$ and $[1, 0]$ for $gender = Male$. The numeric data are first partitioned into buckets, and then we have a binary vector for each bucket, e.g., we can use $[0, 0, 0, 1]$ for child whose $age \in$[0,14], $[0, 0, 1, 0]$ for youth whose $age \in$[15,24], $[0, 1, 0, 0]$ for adult whose $age \in$[25,64], and $[1, 0, 0, 0]$ for seniors whose $age \geq$65.

Since vector $\boldsymbol{x}$ is high-dimensional and very sparse, and different feature fields have various lengths, DRS models usually introduce an embedding layer to transform each binary vector $\boldsymbol{x}_i$ into a low-dimensional continuous vector as:

$$\boldsymbol{e}_i = \boldsymbol{v}_i \boldsymbol{x}_i \tag{6.1}$$

where $\boldsymbol{v}_i \in R^{d \times u_i}$ is the weight matrix with u_i the number of unique feature values in the i^{th} feature field, and d is the pre-defined size of low-dimensional vectors[1]. Finally, the embedding layer will output the concatenation of embedding vectors from all feature fields:

$$\boldsymbol{E} = [\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_m] \tag{6.2}$$

6.2.2.2 Interaction Layer

After representing the input features as low-dimensional embeddings, DRS models usually develop an interaction layer to explicitly capture the interactions among feature fields. The most widely used method is factorization machine (FM) [106]. In addition to the linear interactions among features, FM can explicitly model the pairwise (second-order) feature

[1]For multi-valued features (e.g.,"Interest=Movie, Sports"), the feature embedding is the sum or average of multiple embeddings [28].

interactions via the inner product of feature embeddings:

$$[\langle \boldsymbol{e}_1, \boldsymbol{e}_2 \rangle, \langle \boldsymbol{e}_1, \boldsymbol{e}_3 \rangle, \ldots, \langle \boldsymbol{e}_{m-1}, \boldsymbol{e}_m \rangle] \tag{6.3}$$

where $\langle \cdot, \cdot \rangle$ is the inner product of two embeddings, and the number of pairwise feature interactions is $\boldsymbol{C}_m^2$. Then, the interaction layer will output:

$$l_{fm} = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + \sum_{i=1}^{m} \sum_{j>i}^{m} \langle \boldsymbol{e}_i, \boldsymbol{e}_j \rangle \tag{6.4}$$

Where $\boldsymbol{w}$ is the weight over the binary vector $\boldsymbol{x}$ of input features. The first term represents the impact of first-order feature interactions, and the second term reflects the impact of second-order feature interactions. FM can explicitly model even higher order interactions, such as $\sum_{i=1}^{m} \sum_{j>i}^{m} \sum_{t>j}^{m} \langle \boldsymbol{e}_i, \boldsymbol{e}_j, \boldsymbol{e}_t \rangle$ for third-order, but this will add a lot of computation.

6.2.2.3 MLP Layer

MLP Layer combines and transforms the features, e.g., $\boldsymbol{E}$ and l_{fm}, with several fully-connected layers and activations. The output of each layer is:

$$\boldsymbol{h}_{l+1} = \text{relu}\left(\boldsymbol{W}_l \boldsymbol{h}_l + \boldsymbol{b}_l\right) \tag{6.5}$$

where $\boldsymbol{W}_l$ is the weight matrix and $\boldsymbol{b}_l$ is the bias vector for the l^{th} hidden layer. $\boldsymbol{h}_0$ is the input of first fully-connected layer, and we denote the final output of MLP layer as MLP($\boldsymbol{h}_0$).

6.2.2.4 Output Layer

Finally, the output layer, which is subsequent to the previous layers, will generate the prediction $\hat{\boldsymbol{y}}$ of a user-item interaction data example. The input $\boldsymbol{h}_{out}$ of output layer can be different in different DRS models, e.g., $\boldsymbol{h}_{out} = [l_{fm} + \text{MLP}(\boldsymbol{E})]$ in DeepFM [51] and $\boldsymbol{h}_{out} = \text{MLP}(l_{fm}, \boldsymbol{E})$ in IPNN [103], shown in Figure 6.2. The output layer will yield the prediction $\hat{\boldsymbol{y}}$ of the user-item interaction as:

$$\hat{\boldsymbol{y}} = \sigma\left(\boldsymbol{W}_o \boldsymbol{h}_{out} + \boldsymbol{b}_o\right) \tag{6.6}$$

where $\boldsymbol{W}_o$ and $\boldsymbol{b}_o$ are the weight matrix and bias vector for the output layer. Activation function $\sigma(\cdot)$ is selected based on different recommendation tasks, such as *sigmoid* for binary classification [51], and *softmax* for multi-class classification [128]. Finally, given a set of candidate loss functions, such as mean-squared-error, categorical hinge and cross-entropy, we can compute the candidate losses $\mathcal{L}_{\mathcal{C}}$:

$$\mathcal{L}_{\mathcal{C}} = [\ell_1(\boldsymbol{y}, \hat{\boldsymbol{y}}), \ell_2(\boldsymbol{y}, \hat{\boldsymbol{y}}), \cdots, \ell_n(\boldsymbol{y}, \hat{\boldsymbol{y}})] \tag{6.7}$$

where $\boldsymbol{y}$ is the ground truth label and n is the number of candidate loss functions.

6.2.3 Loss Function Search

AutoLoss aims to adaptively and automatically search the optimal loss function, which can enhance the prediction quality and training efficiency of the DRS network. This is naturally challenging because of the complex relationship between the DRS parameters and the probabilities over candidate loss functions. To address this challenge, many existing

works have focused on developing the fusing strategies for multiple loss functions, which can take advantage of multiple loss functions in a weighted sum manner:

$$\begin{aligned} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}; \boldsymbol{\alpha}) &= \sum_{i=1}^{n} \alpha_i \cdot \ell_i(\boldsymbol{y}, \hat{\boldsymbol{y}}) \\ \text{s.t. } \sum_{i=1}^{n} \alpha_i &= 1, \quad \alpha_i > 0 \; \forall i \in [1, n] \end{aligned} \tag{6.8}$$

where $\boldsymbol{y}$ is the ground truth, $\hat{\boldsymbol{y}}$ is the prediction from DRS network, and ℓ_i is the i^{th} candidate loss function. The continuous loss weights $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \cdots, \alpha_n]$ measure the candidates' contributions in the final loss function $\mathcal{L}$. However, this method relies on exhaustively or manually search of loss weights from a large search space, which is extremely costly. Also, this soft fusing strategy cannot completely eliminate the impact of suboptimal candidate loss functions on the final loss function $\mathcal{L}$, thus, a hard selection method is desired. However, hard selection usually leads to the training framework not end-to-end differentiable.

Reinforcement learning (RL) is a potential solution to tackle the hard selection problem. However, since the RL is generally built upon the Markov decision process, it utilizes temporal-difference to make sequential actions. Consequently, the agent can only receive the reward until the optimal loss function is selected and the DRS is evaluated. In other words, the temporal-difference setting can suffer from delayed rewards. To address this issue, we introduce the Gumbel-softmax operation to simulate the hard selection over candidate loss functions, where the non-differentiable sampling is approximated from a categorical distribution based on a differentiable sampling from the Gumbel-softmax distribution [58].

Given the continuous loss weights $[\alpha_1, \cdots, \alpha_n]$ over candidate loss functions, we can draw

a hard selection z through the Gumbel-max trick [47] as:

$$z = \text{one\_hot}\left(\arg\max_{i\in[1,n]}\left[\log\alpha_i + g_i\right]\right) \tag{6.9}$$

where $g_i = -\log\left(-\log\left(u_i\right)\right)$ and $u_i \sim Uniform(0,1)$. The independent and identically distributed (i.i.d) *gumbel noises* $\{g_i\}$ disturb the $\{\log\alpha_i\}$ terms. Also, they make the arg max operation equivalent to drawing a sample from loss weights $\alpha_1, \cdots, \alpha_n$. However, because of the arg max operation, this sampling method is non-differentiable. We tackle this problem by straight-through Gumbel-softmax [58], which leverages a softmax function as a differentiable approximation to the arg max operation:

$$p_i = \frac{\exp\left(\left(\log\left(\alpha_i\right) + g_i\right)/\tau\right)}{\sum_{j=1}^{n}\exp\left(\left(\log\left(\alpha_j\right) + g_j\right)/\tau\right)}, \quad \forall i \in [1,n] \tag{6.10}$$

where p_i is the probability of selecting the i^{th} candidate loss function. The temperature parameter τ is introduced to manage the smoothness of the Gumbel-softmax operation's output. Specifically, the output approaches a one-hot vector if τ is closer to zero. Then the final loss function $\mathcal{L}$ can be reformulated as:

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}; \boldsymbol{p}) = \sum_{i=1}^{n} p_i \cdot \ell_i(\boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{6.11}$$

In conclusion, the loss function search process becomes end-to-end differentiable by introducing the Gumbel-softmax operation with a similar hard selection performance. Next, we will discuss how to generate data example-level loss weights $[\alpha_1, \cdots, \alpha_n]$.

6.2.4 Controller Network

As in Eq. (6.8), we suppose that $[\alpha_1, \cdots, \alpha_n]$ are the original (continuous) class probabilities over n candidate loss functions before the Gumbel-softmax operation. This assumption aims to learn a set of unified and static probabilities over the candidate loss functions. However, the environment of real-world commercial recommendation platforms is always non-stationary, and different user-item interaction examples have varying convergence behaviors. This cannot be handled by unified and static probabilities, resulting in suboptimal model performance, generalizability and transferability.

We propose a controller network to address this challenge, which learns to generate original class probabilities for each data example. Motivated by curriculum learning [7, 60], the original class probabilities should be generated according to the ground truth labels $\boldsymbol{y}$ and the output of DRS network $\hat{\boldsymbol{y}}$. Therefore, the input of the controller network is a mini-batch $(\boldsymbol{y}, \hat{\boldsymbol{y}})$, followed by the MLP layer with several fully-connected layers like Eq. (6.5). Afterwards, the controller's output layer generates continuous class probabilities $[\alpha_1^b, \cdots, \alpha_n^b]$ $\forall b \in [1, B]$ for each data example in the mini-batch via a standard *softmax* activation, where B is the size of mini-batch. In other word, each data example has individual probabilities. The controller can enhance the recommendation quality, model generalizability and transferability, which is validated by the extensive experiments.

6.2.5 An Optimization Method

In above subsections, we formulate the loss function search as an architectural optimization problem and introduce the Gumbel-softmax that makes the framework end-to-end differentiable. Now, we discuss the optimization for the AutoLoss framework.

In AutoLoss, the parameters to be optimized are from two networks. We denote the main DRS network's parameters as $\boldsymbol{W}$, and the controller network's parameters as $\boldsymbol{V}$. Note that $\boldsymbol{p}$ are directly generated by the Gumbel-softmax operation based on the controller's output $\boldsymbol{\alpha}$ as in Eq. (6.10). Inspired by automated machine learning techniques [100], $\boldsymbol{W}$ and $\boldsymbol{V}$ should not be updated on the same training data batch like traditional supervised learning methods. This is because the optimization of them is highly dependent on each other. As a result, updating $\boldsymbol{W}$ and $\boldsymbol{V}$ on the same training batch can lead to the model over-fitting on the training examples.

According to the end-to-end differentiable property of AutoLoss, we update $\boldsymbol{W}$ and $\boldsymbol{V}$ through gradient descent utilizing the differentiable architecture search (DARTS) techniques [80]. To be specific, $\boldsymbol{W}$ and $\boldsymbol{V}$ are alternately updated on training and validation batches by minimizing the training loss $\mathcal{L}_{train}$ and validation loss $\mathcal{L}_{val}$, respectively. This forms a bi-level optimization problem [100], where controller parameters $\boldsymbol{V}$ and DRS parameters $\boldsymbol{W}$ are considered as the upper- and lower-level variables:

$$\begin{aligned} &\min_{\boldsymbol{V}} \mathcal{L}_{val}\big(\boldsymbol{W}^*(\boldsymbol{V}), \boldsymbol{V}\big) \\ &s.t.\ \boldsymbol{W}^*(\boldsymbol{V}) = \arg\min_{\boldsymbol{W}} \mathcal{L}_{train}(\boldsymbol{W}, \boldsymbol{V}^*) \end{aligned} \tag{6.12}$$

where directly optimizing $\boldsymbol{V}$ thoroughly via Eq. (6.12) is intractable since the inner optimization of $\boldsymbol{W}$ is extremely costly. To tackle this issue, we use an approximation scheme for the inner optimization:

$$\boldsymbol{W}^*(\boldsymbol{V}) \approx \boldsymbol{W} - \xi \nabla_{\boldsymbol{W}} \mathcal{L}_{train}(\boldsymbol{W}, \boldsymbol{V}) \tag{6.13}$$

where ξ is the predefined learning rate. This approximation scheme estimates $\boldsymbol{W}^*(\boldsymbol{V})$ by

Algorithm 6.1: An Optimization Algorithm for AutoLoss via DARTS.

Input: features $\boldsymbol{x}$ and ground-truth labels $\boldsymbol{y}$
Output: well-learned parameters $\boldsymbol{W}^*$ and $\boldsymbol{V}^*$

1: **while** not converged **do**
2: Sample a mini-batch of validation data examples
3: Estimate the approximation of $\boldsymbol{W}^*(\boldsymbol{V})$ via Eq. (6.13)
4: Update $\boldsymbol{V}$ by descending $\nabla_{\boldsymbol{V}} \mathcal{L}_{val}\big(\boldsymbol{W}^*(\boldsymbol{V}), \boldsymbol{V}\big)$
5: Sample a mini-batch of training data examples
6: Update $\boldsymbol{W}$ by descending $\nabla_{\boldsymbol{W}} \mathcal{L}_{train}(\boldsymbol{W}, \boldsymbol{V})$
7: **end while**

descending only one step toward the gradient $\nabla_{\boldsymbol{W}} \mathcal{L}_{train}(\boldsymbol{W}, \boldsymbol{V})$, rather than optimizing $\boldsymbol{W}(\boldsymbol{V})$ thoroughly. To further enhance the computation efficiency, we can set $\xi = 0$, i.e., the first-order approximation.

We detail the AutoLoss optimization via DARTS in Algorithm 6.1. More specifically, in each iteration, we first sample a mini-batch validation data examples of user-item interactions (line 2); next, we estimate (but do not update) $\boldsymbol{W}^*(\boldsymbol{V})$ via the approximation scheme in Eq. (6.13) (line 3); then, we update the controller parameters $\boldsymbol{V}$ by one step based on the estimation (line 4); afterward, we sample a mini-batch training data examples (line 5); and finally, we update the $\boldsymbol{W}$ via descending $\nabla_{\boldsymbol{W}} \mathcal{L}_{train}(\boldsymbol{W}, \boldsymbol{V})$ by one step (line 6).

6.3 Experiment

This section will conduct extensive experiments using various datasets to evaluate the effectiveness of AutoLoss. We first introduce the experimental settings, then compare AutoLoss with representative baselines, and finally conduct model component and transferability analysis.

Table 6.1: Statistics of the datasets.

Data	Criteo	ML-20m
# Interactions	45,840,617	20,000,263
# Feature Fields	39	2
# Feature Values	1,086,810	165,771
# Behavior	click or not	rating 1∼5

6.3.1 Datasets

We evaluate our model on two datasets, including Criteo and ML-20m. Below we introduce these datasets and more statistics about them can be found in Table 6.1.

- **Criteo**[2]: It is a real-world commercial dataset to assess click-through rate prediction models for online ads. It consists of 45 million data examples, i.e., users' click records on displayed ads. Each example contains $m = 39$ anonymous feature fields, where 13 fields are numerical and 26 fields are categorical. 13 numerical fields are converted into categorical features through bucketing.

- **ML-20m**[3]: This is a benchmark dataset to evaluate recommendation algorithms, which contains 20 million users' 5-star ratings on movies. The dataset includes 27,278 movies and 138,493 users, i.e., $m = 2$ feature fields, where each user has at least 20 ratings.

6.3.2 Evaluation Metrics

AutoLoss is general for many recommendation tasks. To evaluate its effectiveness, we conduct *binary classification* (i.e., click-through rate prediction) on Criteo, and *multi-class classification* (i.e., 5-star ratings) on ML-20m. The two classification experiments are evaluated by AUC[4]

[2]https://www.kaggle.com/c/criteo-display-ad-challenge/

[3]https://grouplens.org/datasets/movielens/20m/

[4]We evaluate the AUC for multiclass classification in a one-vs-rest manner.

and Logloss, where higher AUC or lower Logloss mean better performance. It is worth noting that slightly higher AUC and lower Logloss at 0.001-level are considered significant in recommendations [51].

6.3.3 Implementation

We implement AutoLoss based on a public library[5], which contains 16 representative recommendation models. We develop AutoLoss as an independent class, so we can easily apply our framework for all these models. In this chapter, we only show the results on DeepFM [51] and IPNN [103] due to the page limitation. To be specific, AutoLoss framework mainly contains two networks, i.e., the DRS network and the controller network.

For the DRS network, (a) *Embedding layer*: we set the embedding size as 16 following the existing works [169]. (b) *Interaction layer*: we leverage factorization machine and inner product network to capture the interactions among feature fields for DeepFM and IPNN, respectively. (c) *MLP layer*: we have two fully-connected layers, and the layer size is 128. We also employ batch normalization, dropout ($rate = 0.2$) and ReLU activation for both layers. (d) *Output layer*: original DeepFM and IPNN are designed for click-through rate prediction, which use *sigmoid* activation for negative log-likelihood function. To fit the 5-class classification task on ML-20m, we modify the output layer correspondingly. i.e., the output layer is 5-dimensional with *softmax* activation.

For the controller network, (a) *Input layer*: the inputs are the ground truth labels $\boldsymbol{y}$ and the predictions $\hat{\boldsymbol{y}}$ from DRS network. (b) *MLP layer*: we also use two fully-connected layers with the layer size 128, batch normalization, dropout ($rate = 0.2$) and ReLU activation. (3) *Output layer*: the controller network will output continuous loss probabilities $\boldsymbol{\alpha}$ with *softmax*

[5] https://github.com/rixwew/pytorch-fm

activation, whose dimension equals to the number of candidate loss functions.

For other hyper-parameters, (a) *Gumbel-softmax*: we use an annealing scheme for temperature $\tau = \max(0.01, 1 - 0.00005 \cdot t)$, where t is the training step. (b) *Optimization*: we set the learning rate as 0.001 for updating both DRS network and controller network with Adam optimizer and batch-size 2000. (c) We select the hyper-parameters of the AutoLoss framework via cross-validation, and we also do parameter-tuning for baselines correspondingly for a fair comparison.

6.3.4 Overall Performance Comparison

AutoLoss is compared with the following loss function design and search methods:

- Fixed loss function: the first group of baselines leverages a predefined and fixed loss function. We utilize Focal loss, KL divergence, Hinge loss and cross-entropy (CE) loss for both classification tasks.

- Fixed weights over loss functions: this group of baselines aims to learn fixed weights over the loss functions in the first group, without considering the difference among data examples. In this group, we use the meta-learning method MeLU [68], as well as automated machine learning methods BOHB [34] and DARTS [80].

- Data example-wise loss weights: this group learns to assign different loss weights for different data examples according to their convergence behaviors. One existing work, stochastic loss function (SLF) [82], belongs to this group.

The overall performance is shown in Table 6.2. It can be observed that:

- The first group of baselines achieves the worst recommendation performance in both recommendation tasks. Their optimizations are based on predefined and fixed loss functions

Table 6.2: Performance comparison of different loss function search methods.

Dataset	Model	Metric	Methods								
			Focal	KL	Hinge	CE	MeLU	BOHB	DARTS	SLF	AutoLoss
Criteo	DeepFM	AUC	0.8046	0.8042	0.8049	0.8056	0.8063	0.8065	0.8067	0.8081	**0.8092***
		Logloss	0.4466	0.4469	0.4463	0.4457	0.4436	0.4435	0.4433	0.4426	**0.4416***
Criteo	IPNN	AUC	0.8077	0.8072	0.8079	0.8085	0.8090	0.8092	0.8093	0.8098	**0.8108***
		Logloss	0.4435	0.4437	0.4432	0.4428	0.4423	0.4422	0.4423	0.4418	**0.4409***
ML-20m	DeepFM	AUC	0.7681	0.7682	0.7685	0.7692	0.7695	0.7695	0.7696	0.7705	**0.7717***
		Logloss	1.2320	1.2317	1.2316	1.2310	1.2307	1.2305	1.2305	1.2299	**1.2288***
ML-20m	IPNN	AUC	0.7721	0.7722	0.7725	0.7733	0.7735	0.7734	0.7736	0.7745	**0.7756***
		Logloss	1.2270	1.2269	1.2266	1.2260	1.2256	1.2257	1.2255	1.2249	**1.2236***

"*" indicates the statistically significant improvements (i.e., two-sided t-test with $p < 0.05$) over the best baseline.

during the training stage. This result demonstrates that leveraging a predefined and fixed loss function throughout can downgrade the recommendation quality.

- The methods in the second group outperform those in the first group. These methods try to learn weights over candidate loss functions according to their contributions to the optimization, and then combine them in a weighted sum manner. This validates that incorporating multiple loss functions in optimization can enhance the performance of deep recommender systems.

- The second group performs worse than the SLF, since the weights they learned are unified and static, which completely overlooks the various behaviors among different data examples. Therefore, SLF performs better by taking this factor into account.

- The decision network of SLF is optimized on the same training batch with the main DRS network via back-propagation, which can lead to over-fitting on the training batch. AutoLoss updates the DRS network on the training batch while updating the controller on the validation batch, which improves the model generalizability and results in better recommendation performance.

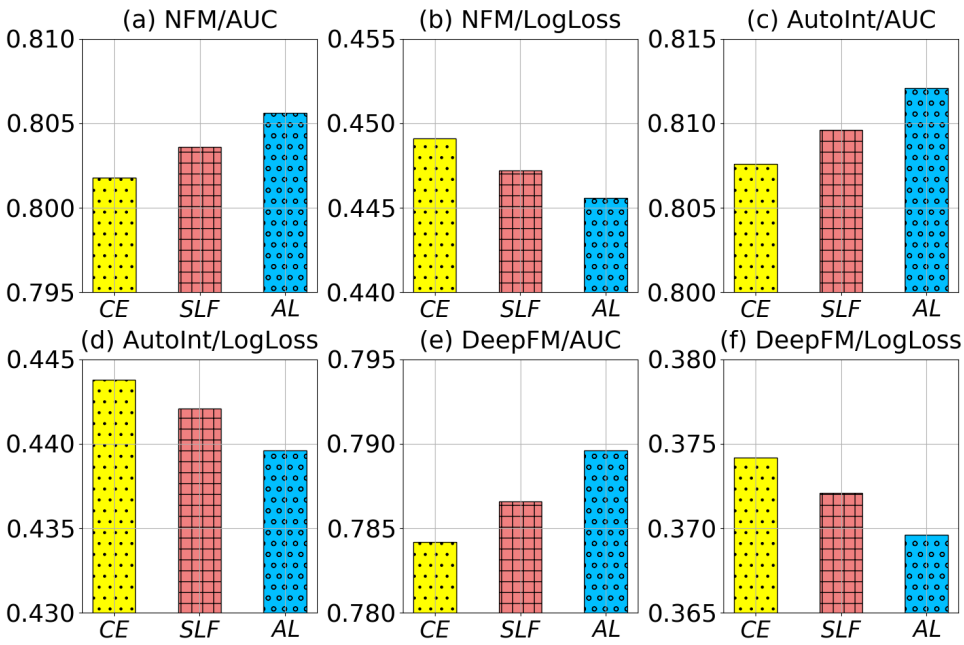

Figure 6.3: Transferability study results.

To summarize, AutoLoss achieves significantly better performance than state-of-the-art baselines on both datasets and tasks, which demonstrates its effectiveness.

6.3.5 Transferability Study

In this subsection, we study the transferability of the controller. Specifically, we want to investigate (i) whether the controller trained with one DRS model can be applied to other DRS models; and (ii) whether the controller learned on one dataset can be directly used on other datasets.

- To study the transferability across different DRS models, we leverage the controller trained via DeepFM and AutoLoss on Criteo, fix its parameters and apply it to train NFM [54] and AutoInt [120] on Criteo. The results are demonstrated in Figure 6.3 (a)-(d), where (i) "CE" means that we directly train the new DRS model via minimizing the cross-entropy (CE) loss, which is the best single and fixed loss function in Table 6.2; (ii) "SLF" is that we

use the controller upon DeepFM and SLF, which is the best baseline in Table 6.2; and (iii) "AL" denotes that we use the controller based on DeepFM and AutoLoss. From the figures, we can observe that SLF performs superior to CE, which indicates that a pre-trained controller can improve other DRS models' training performance. More importantly, AL outperforms SLF, which validates AutoLoss's better transferability across different DRS models.

- To study the transferability between different datasets, we train a controller upon Criteo dataset with DeepFM and AutoLoss, fix its parameters and apply it to train a new DeepFM on the Avazu dataset[6], i.e., "AL". Also, we denote that (i) "CE": DeepFM is directly optimized by minimizing cross-entropy (CE) loss on Avazu dataset; and (ii) "SLF": DeepFM is optimized on the new dataset with the assistance of a controller pre-trained with DeepFM+SLF on Criteo. In Figure 6.3 (e)-(f), AL shows superior performance over CE and SLF, which proves its better transferability between different datasets.

In summary, AutoLoss has better transferability across different DRS models and different recommendation datasets, which improves its usability in real-world recommender systems.

6.3.6 Impact of Model Components

In this subsection, in order to understand the contributions of important model components of AutoLoss, we systematically eliminate each component and define the following variants:

- **AL-1**: This variant aims to assess the contribution of the controller. Thus, we assign equivalent weights on four candidate loss functions, i.e., $[0.25, 0.25, 0.25, 0.25]$.

[6] Avazu is another benchmark dataset for CTR prediction, which contains 40 million user clicking behaviors in 11 days with $M = 22$ categorical feature fields. https://www.kaggle.com/c/avazu-ctr-prediction/

Table 6.3: Impact of model components.

Dataset	Model	Metric	Methods		
			AL-1	AL-2	AutoLoss
Criteo	DeepFM	AUC ↑	0.8052	0.8083	**0.8092***
		Logloss ↓	0.4460	0.4422	**0.4416***
Criteo	IPNN	AUC ↑	0.8081	0.8102	**0.8108***
		Logloss ↓	0.4431	0.4416	**0.4409***

"*" indicates the statistically significant improvements (i.e., two-sided t-test with $p < 0.05$) over the best baseline. ↑: the higher the better; ↓: the lower the better.

- **AL-2**: In this variant, we eliminate the Gumbel-softmax operation, and directly use the controller's output, i.e., the continuous loss probabilities $\boldsymbol{\alpha}$ from standard *softmax* activation, which aims to evaluate the impact of Gumbel-softmax.

The results on the Criteo dataset are shown in Table 6.3:

- First, AL-1 has worse performance than AutoLoss, which validates the necessity to introduce the controller network. It is noteworthy that, AL-1 performs worse than all loss function search methods, and even the fixed cross-entropy (CE) loss in Table 6.2, which indicates that equally incorporating all candidate loss functions cannot guarantee better performance.

- Second, AutoLoss outperforms AL-2. The main reason is that AL-2 always generates gradients based on all the loss functions, which introduces some noisy gradients from the suboptimal candidate loss functions. In contrast, AutoLoss can obtain appropriate gradients by filtering out suboptimal loss functions via Gumbel-softmax, which enhances the model robustness.

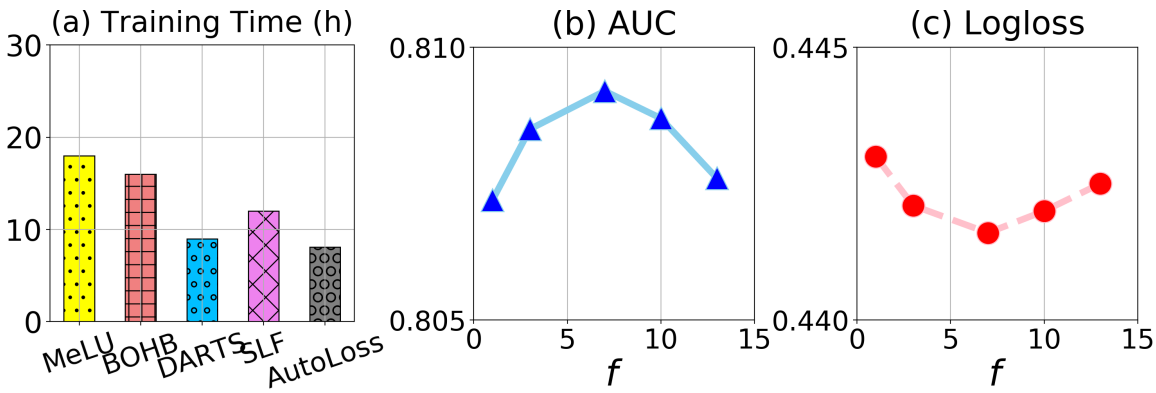

Figure 6.4: Efficiency study results.

6.3.7 Efficiency Study

This section compares AutoLoss's training efficiency with other loss function searching methods, which is an important metric to deploy a DRS model in real-world applications. Our experiments are based on one GeForce GTX 1060 GPU.

The results of DeepFM on Criteo dataset are illustrated in Figure 6.4 (a). We can observe that AutoLoss achieves the fastest training speed. The reasons are two-fold.

- First, AutoLoss can generate the most appropriate gradients to update DRS, which increases the optimization efficiency.
- Second, we update the controller once after every 7 times DRS is updated, i.e., the controller updating frequency $f = 7$. This trick not only reduces the training time ($\sim 60\%$ in this case) with fewer computations, but also enhances the performance. In Figure 6.4 (b)-(c) where x-axis is f, we find that DeepFM performs the best when $f = 7$, while updating too frequently/infrequently can lead to suboptimal AUC/Logloss.

To summarize, AutoLoss can efficiently achieve better performance, making it easier to be launched in real-world recommender systems.

6.4 Related Work

In this section, we briefly introduce the works related to our study. We first go over the latest studies in loss function search and then review works about AutoML for recommendations.

Loss Function Search. The loss function plays an essential part in a deep learning framework. The choice of the loss function significantly affects the performance of the learned model. A lot of efforts have been made to design desirable loss functions for specific tasks. For example, in the field of image processing, Rahman et al. [104] argued that the typical cross-entropy loss for semantic segmentation shows great limitations in aligning with evaluation metrics other than global accuracy. Ronneberger et. al [110, 143] designed loss functions by taking class frequency into consideration to cater to the mIoU metric. Caliva et al. [16, 102] designed losses with larger weights at boundary regions to improve the boundary F1 score. Liu et al. [84] proposed to replace the traditional Softmax loss with large margin Softmax (L-Softmax) loss to improve feature discrimination in classification tasks. Fan et al. [36] used sphere Softmax loss for the person re-identification task and obtained state-of-the-art results. The loss functions mentioned above are all designed manually, requiring ample expert knowledge, non-trivial time, and many human efforts.

Recently, automated loss function search draws increasing interests of researchers from various machine learning (ML) fields. Xu et al. [146] investigated how to automatically schedule iterative and alternate optimization processes for ML models. A meta-learning framework was proposed to adaptively determine which loss function to use and which parameters to update at each optimization step. Liu et al. [82] proposed to optimize the stochastic loss function (SLF), where the loss function of an ML model was dynamically selected. The loss function selection is determined by loss parameters, including a selective

binary code and a weighting coefficient. During training, model parameters and the loss parameters are learned jointly. Li et al. [71] proposed automatically searching specific surrogate losses to improve different evaluation metrics in the image semantic segmentation task. Besides, Li et al. [70, 135] designed search spaces for a series of existing loss functions and developed algorithms to search for the best parameters of the probability distribution for sampling loss functions. However, their methods are designed exclusively for cross-entropy loss and its variants, making their methods not applicable in our tasks.

AutoML for Recommendation. AutoML techniques are now widely used to automatically design deep recommendation systems. Previous works mainly focused on the design of the embedding layer and the selection of feature interaction patterns.

In terms of the embedding layer, Joglekar et al. [62, 157, 83] proposed novel methods to automatically select the best embedding size for different feature fields in a recommendation system. Zhao et al. [158, 81] proposed to dynamically search embedding sizes for users and items based on their popularity in the streaming setting. Similarly, Ginart et al. [44] proposed to use mixed dimension embeddings for users and items based on their query frequency. Kang et al. [63] proposed a multi-granular quantized embeddings (MGQE) technique to learn impact embeddings for infrequent items. Cheng et al. [25] proposed to perform embedding dimension selection with a soft selection layer, making the dimension selection more flexible. Guo et al. [50] focused on the embeddings of numerical features. They proposed AutoDis, which automatically discretizes features in numerical fields and maps the resulting categorical features into embeddings.

As for feature interaction, Luo et al. [88] proposed AutoCross that produces high-order cross features by performing beam search in a tree-structure feature space. Song et. al [119, 64, 78, 149] proposed to automatically discover feature interaction architectures for

click-through rate (CTR) prediction. Tsang et al. [130] proposed a method to interpret the feature interactions from a source recommendation model and apply them in a target recommendation model.

Chapter 7

Conclusions

7.1 Dissertation Summary

In this dissertation, we have described our efforts devoted to adaptive and automated deep recommender systems. Particularly, we have discussed our proposed (i) adaptive recommendation policies that continuously update recommendation strategies during the interactions and maximize the expected long-term cumulative reward from users, and (ii) effective approaches that design deep recommender system frameworks from an automated and data-driven manner.

In chapter 2, we propose a novel page-wise recommendation framework DeepPage, which leverages Deep Reinforcement Learning to automatically learn the optimal recommendation strategies and optimizes a page of items simultaneously. Reinforcement learning based recommender systems have two advantages: (1) they can continuously update strategies during the interactions, and (2) they are able to learn a strategy that maximizes the long-term cumulative reward from users. Unlike previous work, we propose a novel Actor-Critic framework, which can be applied in scenarios with large and dynamic item space and reduce redundant computation significantly. Furthermore, the proposed framework can capture the real-time preference and optimize a page of items jointly, which can boost recommendation performance. We evaluate our framework with extensive experiments based on data from a real

e-commerce company. The results show that (1) DeepPage can improve the recommendation performance; and (2) capturing users' real-time preference and jointly optimizing a page is helpful for accurate recommendation.

In chapter 3, we propose a two-level deep reinforcement learning framework RAM with novel Deep Q-network architectures for the mixed display of recommendation and advertisements in online recommender systems. Upon a user's request, the RS (i.e., first level) first recommends a list of items based on user's historical behaviors, then the AS (i.e., second level) inserts ads into the rec-list, which can make three decisions, i.e., whether to insert an ad into the rec-list; and if yes, the AS will select the optimal ad and insert it into the optimal location. The proposed two-level framework aims to simultaneously optimize the long-term user experience and immediate advertising revenue. It is worth noting that the proposed AS architecture can take advantage of two conventional DQN architectures, which can evaluate the Q-value of two kinds of related actions simultaneously. We evaluate our framework with extensive experiments based on data from a short video site TikTok. The results show that our framework can jointly improve recommendation and advertising performance.

In chapter 4, we propose a novel user simulator, UserSim, based on Generative Adversarial Network (GAN) framework, which models real users' behaviors from users' historical logs, and tackle the two challenges: (i) the recommended item distribution is complex within users' historical logs, and (ii) labeled training data from each user is limited. The GAN-based user simulator can naturally resolve these two challenges and can be used to pre-train and evaluate new recommendation algorithms before launching them online. To be specific, the generator captures the underlying item distribution of users' historical logs and generates indistinguishable fake logs that can be used as augmentations of real logs. The discriminator can predict users' feedback of a recommended item based on users' browsing logs, taking

advantage of both supervised and unsupervised learning techniques. In order to validate the effectiveness of the proposed user simulator, we conduct extensive experiments based on benchmark datasets. The results show that the proposed user simulator can improve the user behavior prediction performance in recommendation tasks over representative baselines.

In chapter 5, we propose a novel framework AutoDim, which targets automatically assigning different embedding dimensions to different feature fields in a data-driven manner. In real-world recommender systems, due to the huge amounts of feature fields and the highly complex relationships among embedding dimensions, feature distributions and neural network architectures, it is difficult, if possible, to manually allocate different dimensions to different feature fields. Thus, we proposed an AutoML based framework to automatically select from different embedding dimensions. To be specific, we first provide an end-to-end differentiable model, which computes the weights over different dimensions for different feature fields simultaneously in a soft and continuous form, and we propose an AutoML-based optimization algorithm; then according to the maximal weights, we derive a discrete embedding architecture, and re-train the DLRS parameters. We evaluate the AutoDim framework with extensive experiments based on widely used benchmark datasets. The results show that our framework can maintain or achieve slightly better performance with much fewer embedding space demands.

In chapter 6, we propose a novel end-to-end framework, AutoLoss, to enhance recommendation performance and deep recommender systems' training efficiency by selecting appropriate loss functions in a data-driven manner. AutoLoss can automatically select the proper loss function for each data example according to their varied convergence behaviors. To be specific, we first develop a novel controller network, which generates continuous loss weights based on the ground truth labels and the DRS' predictions. Then, we introduce a

Gumbel-softmax operation to simulate the hard selection over candidate loss functions, which filters out the noisy gradients from suboptimal candidates. Finally, we can select the optimal candidate according to the output from Gumbel-softmax. We conduct extensive experiments to validate the effectiveness of AutoLoss on two widely used benchmark datasets. The results show that our framework can improve recommendation performance and training efficiency with excellent transferability.

7.2 Future Works

Given the promising achievements from our works, we believe more dedicated efforts should be devoted to the policies and framework of deep recommender systems. With respect to future work, we are interested in investigating the following directions:

- **Off-policy or Offline RL in Recommendations**: Due to the system architecture in modern web services, it is almost impossible to perform online reinforcement learning in real-world recommendations [79, 175, 172]. In practice, the ranking functions (or policies) are trained periodically, e.g., daily or weekly, based on the recently collected log data. As such, the ways of training of any DRL-based policies or value functions can only be off-policy or offline. Due to the nature of RL, once the ranking policy is updated, the corresponding data distribution, i.e., occupancy measure, will change, which will make the training biased. In DRL, off-policy methods restrict the new policy around the data-collecting policy, while offline RL methods usually constrain the learning policy to support the logged data. So far, there is barely no established research addressing such a problem in recommendations.

- **Formulating ranking as a decision-making task**: Ranking is a process that takes a query as input and outputs the ranking of candidate items. Different formulations of

ranking lead to different inductive biases, which may fit different recommendation scenarios. For example, sequentially decide to pick the next-slot item from the remaining candidates fits more to the sequential recommendation tasks but may suffer from efficiency issue [19], while learning a direct ranking function may fail to capture the user's specific context when browsing the ranked list [32, 61, 13]. To our knowledge, there has not yet been a deep investigation of the effectiveness of formulating ranking as a decision-making task.

- **Dynamics Simulation for RL Recommendations**: As previously mentioned, a user's behavior when facing a ranked item list can be regarded as a dynamic process, which can be formulated as a prediction or generation process called user click model [9, 29]. Learning such a user click model (i.e., the dynamics for RL-based ranking policies) is still challenging since the logged data is always biased, e.g., position bias and selection bias. Moreover, how well the learned user click model will guide the training of the RL-based ranking policies is non-guaranteed, because the common problem of compounding error of the learned dynamics model for simulation data generation [2]. Meanwhile, there is no commonly recognized simulation benchmark for DRL-based ranking or recommendation tasks, although some attempts have been performed, such as UserSim in Chapter 4.

- **Sample Efficiency for RL Recommendations**: Sample efficiency has been of key problem for DRL since most DRL methods are model-free. Given sufficient training data from the agent interacting with the environment, the paradigm of model-free RL is suitable for training deep neural network based value functions or policies. However, most recommender systems directly interact with the users and collect the training data, which is normally insufficient to train the model-free RL solutions.

- **Sparse & Biased Feedback Data**: Feedback data is commonly biased. In RL view,

the experience data is sampled from the occupancy measure distribution of the policy-environment interaction. Although off-policy training methods can still help improve the policy based on biased data, the sample efficiency is seriously reduced. Moreover, in some information retrieval scenarios such as online advertising, the positive reward is highly sparse (e.g., 0.3% click-through rate for display ads), which results in very low efficiency or failure of RL.

- **Online Deployment of RL Solutions**: From industry perspective, deploying DRL solutions onto production recommender systems is challenging. The common model pipelines in recommender systems center on relevance estimation models, e.g., relevance or CTR estimation, while the DRL model pipelines center on the policy module. Bridging gap between two generations of model pipelines should be positioned as high priority for applied research teams in this field.

- **Fairness in Recommendations**: As recommender systems influence more and more people in their daily life, the problem of fairness in recommendation is becoming more and more important. Most of the prior approaches to fairness-aware recommendation have been situated in a static or one-shot setting, where the protected groups of items are fixed, and the model provides a one-time fairness solution based on fairness-constrained optimization. This fails to consider the dynamic nature of the recommender systems, where attributes such as item popularity may change over time due to the recommendation policy and user engagement. For example, products that were once popular may become no longer popular, and vice versa. As a result, the system that aims to maintain long-term fairness on the item exposure in different popularity groups must accommodate this change in a timely fashion. I plan to explore long-term fairness in recommendation and accomplish the

problem through dynamic fairness learning.

- **Feature Interaction in Recommendations**: For recommendation tasks, explicit feature interactions can significantly improve the performance of models. Factorization Models, such as FM, DeepFM, have been proposed to explore the explicit feature interactions. These factorization models, are widely used in various industrial recommender systems. However, all these models are simply either enumerating all feature interactions or requiring human efforts to identify important feature interactions. The former, enumerating all feature interactions, always brings large memory and computation cost to the model and is difficult to be extended into high-order interactions. Besides, useless interactions may bring unnecessary noise and complicate the training process. The latter, human identify important feature interactions, is of high labor cost and risks missing some counter-intuitive (but important) interactions. I plan to conduct research on automating feature interaction.

- **Full Automation in Recommendations**: Most existing AutoML works in recommendations focus on architecture search of their deep neural networks, such as embedding layer [158, 81, 157, 83] and interaction layer [119, 64, 78, 149]. But actually, AutoML can be incorporated to more tasks when we build a deep recommender system, such as the loss function search framework AutoLoss in Chapter 6. In the future, we can study more such applications, such as feature engineering, model selection, optimization and evaluation. Eventually, we aim to provide an end-to-end automatic solution to build recommender systems without human efforts.

BIBLIOGRAPHY

BIBLIOGRAPHY

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

[2] Kavosh Asadi, Dipendra Misra, and Michael Littman. Lipschitz continuity in model-based reinforcement learning. In *International Conference on Machine Learning*, pages 264–273, 2018.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[4] Xueying Bai, Jian Guan, and Hongning Wang. A model-based reinforcement learning with adversarial training for online recommendation. In *Advances in Neural Information Processing Systems*, pages 10735–10746, 2019.

[5] Jie Bao, Yu Zheng, David Wilkie, and Mohamed Mokbel. Recommendations in location-based social networks: a survey. *Geoinformatica*, 19(3):525–565, 2015.

[6] Richard Bellman. *Dynamic programming*. Courier Corporation, 2013.

[7] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[8] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[9] Alexey Borisov, Ilya Markov, Maarten De Rijke, and Pavel Serdyukov. A neural click model for web search. In *Proceedings of the 25th International Conference on World Wide Web*, pages 531–541, 2016.

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

[11] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

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

[13] Han Cai, Kan Ren, Weinan Zhang, Kleanthis Malialis, Jun Wang, Yong Yu, and Defeng Guo. Real-time bidding by reinforcement learning in display advertising. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 661–670. ACM, 2017.

[14] Qingpeng Cai, Aris Filos-Ratsikas, Pingzhong Tang, and Yiwei Zhang. Reinforcement mechanism design for e-commerce. In *Proceedings of the 2018 World Wide Web Conference*, pages 1339–1348. International World Wide Web Conferences Steering Committee, 2018.

[15] Sylvain Calinon, Florent D'halluin, Eric L Sauser, Darwin G Caldwell, and Aude G Billard. Learning and reproduction of gestures by imitation. *IEEE Robotics & Automation Magazine*, 17(2):44–54, 2010.

[16] Francesco Caliva, Claudia Iriondo, Alejandro Morales Martinez, Sharmila Majumdar, and Valentina Pedoia. Distance map loss penalty term for semantic segmentation. *arXiv preprint arXiv:1908.03679*, 2019.

[17] Samprit Chatterjee and Ali S Hadi. *Regression analysis by example*. John Wiley & Sons, 2015.

[18] Haokun Chen, Xinyi Dai, Han Cai, Weinan Zhang, Xuejian Wang, Ruiming Tang, Yuzhou Zhang, and Yong Yu. Large-scale interactive recommendation with tree-structured policy gradient. *arXiv preprint arXiv:1811.05869*, 2018.

[19] Haokun Chen, Xinyi Dai, Han Cai, Weinan Zhang, Xuejian Wang, Ruiming Tang, Yuzhou Zhang, and Yong Yu. Large-scale interactive recommendation with tree-structured policy gradient. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3312–3320, 2019.

[20] Minmin Chen, Alex Beutel, Paul Covington, Sagar Jain, Francois Belletti, and Ed H Chi. Top-k off-policy correction for a reinforce recommender system. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pages 456–464. ACM, 2019.

[21] Shi-Yong Chen, Yang Yu, Qing Da, Jun Tan, Hai-Kuan Huang, and Hai-Hong Tang. Stabilizing reinforcement learning in dynamic environment with application to online recommendation. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1187–1196. ACM, 2018.

[22] Ting Chen, Lala Li, and Yizhou Sun. Differentiable product quantization for end-to-end embedding compression. *arXiv preprint arXiv:1908.09756*, 2019.

[23] Xinshi Chen, Shuang Li, Hui Li, Shaohua Jiang, Yuan Qi, and Le Song. Generative adversarial user model for reinforcement learning based recommendation system. In *International Conference on Machine Learning*, pages 1052–1061, 2019.

[24] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa 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.

[25] Weiyu Cheng, Yanyan Shen, and Linpeng Huang. Differentiable neural input search for recommender systems. *arXiv preprint arXiv:2006.04466*, 2020.

[26] Sungwoon Choi, Heonseok Ha, Uiwon Hwang, Chanju Kim, Jung-Woo Ha, and Sungroh Yoon. Reinforcement learning based recommender system using biclustering technique. *arXiv preprint arXiv:1801.05532*, 2018.

[27] Corinna Cortes and Mehryar Mohri. Auc optimization vs. error rate minimization. In *Advances in neural information processing systems*, pages 313–320, 2004.

[28] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

[29] Xinyi Dai, Jianghao Lin, Weinan Zhang, Shuai Li, Weiwen Liu, Ruiming Tang, Xiuqiang He, Jianye Hao, Jun Wang, and Yong Yu. An adversarial imitation click model for information retrieval. In *Proceedings of the 30th International Conference on World Wide Web*, 2021.

[30] Thomas Degris, Martha White, and Richard S Sutton. Off-policy actor-critic. *arXiv preprint arXiv:1205.4839*, 2012.

[31] Wenkui Ding, Tao Qin, Xu-Dong Zhang, and Tie-Yan Liu. Multi-armed bandit with budget constraint and variable costs. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[32] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

[33] Ludwig Fahrmeir, Thomas Kneib, Stefan Lang, and Brian Marx. *Regression*. Springer, 2007.

[34] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, pages 1437–1446. PMLR, 2018.

[35] Wenqi Fan, Tyler Derr, Xiangyu Zhao, Yao Ma, Hui Liu, Jianping Wang, Jiliang Tang, and Qing Li. Attacking black-box recommendations via copying cross-domain user profiles. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1583–1594. IEEE, 2021.

[36] Xing Fan, Wei Jiang, Hao Luo, and Mengjuan Fei. Spherereid: Deep hypersphere manifold embedding for person re-identification. *Journal of Visual Communication and Image Representation*, 60:51–58, 2019.

[37] Mamdouh Farouk. Measuring sentences similarity: a survey. *arXiv preprint arXiv:1910.03940*, 2019.

[38] Jun Feng, Heng Li, Minlie Huang, Shichen Liu, Wenwu Ou, Zhirong Wang, and Xiaoyan Zhu. Learning to collaborate: Multi-scenario ranking via multi-agent reinforcement learning. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 1939–1948. International World Wide Web Conferences Steering Committee, 2018.

[39] Jim Gao. Machine learning applications for data center optimization. 2014.

[40] Weihao Gao, Xiangjun Fan, Jiankai Sun, Kai Jia, Wenzhi Xiao, Chong Wang, and Xiaobing Liu. Deep retrieval: An end-to-end learnable structure model for large-scale recommendations. *arXiv preprint arXiv:2007.07203*, 2020.

[41] Margherita Gasparini, Alessandro Nuara, Francesco Trovò, Nicola Gatti, and Marcello Restelli. Targeting optimization for internet advertising by learning from logged bandit feedback. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018.

[42] Yingqiang Ge, Shuchang Liu, Ruoyuan Gao, Yikun Xian, Yunqi Li, Xiangyu Zhao, Changhua Pei, Fei Sun, Junfeng Ge, Wenwu Ou, et al. Towards long-term fairness in recommendation. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*, pages 445–453, 2021.

[43] Alexandre Gilotte, Clément Calauzènes, Thomas Nedelec, Alexandre Abraham, and Simon Dollé. Offline a/b testing for recommender systems. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pages 198–206. ACM, 2018.

[44] Antonio Ginart, Maxim Naumov, Dheevatsa Mudigere, Jiyan Yang, and James Zou. Mixed dimension embeddings with application to memory-efficient recommendation systems. *arXiv preprint arXiv:1909.11810*, 2019.

[45] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[46] Jie Gui, Zhenan Sun, Yonggang Wen, Dacheng Tao, and Jieping Ye. A review on generative adversarial networks: Algorithms, theory, and applications. *arXiv preprint arXiv:2001.06937*, 2020.

[47] Emil Julius Gumbel. *Statistical theory of extreme values and some practical applications: a series of lectures*, volume 33. US Government Printing Office, 1948.

[48] Asela Gunawardana and Guy Shani. A survey of accuracy evaluation metrics of recommendation tasks. *Journal of Machine Learning Research*, 10(Dec):2935–2962, 2009.

[49] Hao Guo, Xin Li, Ming He, Xiangyu Zhao, Guiquan Liu, and Guandong Xu. Cosolorec: Joint factor model with content, social, location for heterogeneous point-of-interest recommendation. In *International Conference on Knowledge Science, Engineering and Management*, pages 613–627. Springer, 2016.

[50] Huifeng Guo, Bo Chen, Ruiming Tang, Zhenguo Li, and Xiuqiang He. Autodis: Automatic discretization for embedding numerical features in ctr prediction. *arXiv preprint arXiv:2012.08986*, 2020.

[51] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. Deepfm: a factorization-machine based neural network for ctr prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 1725–1731, 2017.

[52] Lei Guo, Hongzhi Yin, Qinyong Wang, Tong Chen, Alexander Zhou, and Nguyen Quoc Viet Hung. Streaming session-based recommendation. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1569–1577, 2019.

[53] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*, 2015.

[54] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017.

[55] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. Session-based recommendations with recurrent neural networks. In *International Conference on Learning Representations (ICLR)*, 2016.

[56] Eugene Ie, Chih-wei Hsu, Martin Mladenov, Vihan Jain, Sanmit Narvekar, Jing Wang, Rui Wu, and Craig Boutilier. Recsim: A configurable simulation platform for recommender systems. *arXiv preprint arXiv:1909.04847*, 2019.

[57] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[58] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

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

[60] Lu Jiang, Deyu Meng, Shoou-I Yu, Zhenzhong Lan, Shiguang Shan, and Alexander Hauptmann. Self-paced learning with diversity. In *Advances in Neural Information Processing Systems*, pages 2078–2086, 2014.

[61] Junqi Jin, Chengru Song, Han Li, Kun Gai, Jun Wang, and Weinan Zhang. Real-time bidding with multi-agent reinforcement learning in display advertising. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 2193–2201. ACM, 2018.

[62] Manas R Joglekar, Cong Li, Mei Chen, Taibai Xu, Xiaoming Wang, Jay K Adams, Pranav Khaitan, Jiahui Liu, and Quoc V Le. Neural input search for large scale recommendation models. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2387–2397, 2020.

[63] Wang-Cheng Kang, Derek Zhiyuan Cheng, Ting Chen, Xinyang Yi, Dong Lin, Lichan Hong, and Ed H Chi. Learning multi-granular quantized embeddings for large-vocab categorical features in recommender systems. In *Companion Proceedings of the Web Conference 2020*, pages 562–566, 2020.

[64] Farhan Khawar, Xu Hang, Ruiming Tang, Bin Liu, Zhenguo Li, and Xiuqiang He. Autofeature: Searching for feature interactions and their architectures for click-through rate prediction. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 625–634, 2020.

[65] Alexander Kirillov, Ross Girshick, Kaiming He, and Piotr Dollár. Panoptic feature pyramid networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6399–6408, 2019.

[66] Ron Kohavi and Roger Longbotham. Online controlled experiments and a/b testing. *Encyclopedia of machine learning and data mining*, 7(8):922–929, 2017.

[67] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[68] Hoyeop Lee, Jinbae Im, Seongwon Jang, Hyunsouk Cho, and Sehee Chung. Melu: meta-learned user preference estimator for cold-start recommendation. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1073–1082, 2019.

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

[70] Chuming Li, Xin Yuan, Chen Lin, Minghao Guo, Wei Wu, Junjie Yan, and Wanli Ouyang. Am-lfs: Automl for loss function search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 8410–8419, 2019.

[71] Hao Li, Chenxin Tao, Xizhou Zhu, Xiaogang Wang, Gao Huang, and Jifeng Dai. Auto seg-loss: Searching metric surrogates for semantic segmentation. *arXiv preprint arXiv:2010.07930*, 2020.

[72] Lihong Li, Wei Chu, John Langford, Taesup Moon, and Xuanhui Wang. An unbiased offline evaluation of contextual bandit algorithms with generalized linear models. In *Proceedings of the Workshop on On-line Trading of Exploration and Exploitation 2*, pages 19–36, 2012.

[73] Lihong Li, Jin Young Kim, and Imed Zitouni. Toward predicting the outcome of an a/b experiment for search relevance. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, pages 37–46. ACM, 2015.

[74] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1754–1763, 2018.

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

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

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

[78] Bin Liu, Chenxu Zhu, Guilin Li, Weinan Zhang, Jincai Lai, Ruiming Tang, Xiuqiang He, Zhenguo Li, and Yong Yu. Autofis: Automatic feature interaction selection in factorization models for click-through rate prediction. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2636–2645, 2020.

[79] Feng Liu, Ruiming Tang, Xutao Li, Weinan Zhang, Yunming Ye, Haokun Chen, Huifeng Guo, and Yuzhou Zhang. Deep reinforcement learning based recommendation with explicit user-item interactions modeling. *arXiv preprint arXiv:1810.12027*, 2018.

[80] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[81] Haochen Liu, Xiangyu Zhao, Chong Wang, Xiaobing Liu, and Jiliang Tang. Automated embedding size search in deep recommender systems. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2307–2316, 2020.

[82] Qingliang Liu and Jinmei Lai. Stochastic loss function. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 4884–4891, 2020.

[83] Siyi Liu, Chen Gao, Yihong Chen, Depeng Jin, and Yong Li. Learnable embedding sizes for recommender systems. *arXiv preprint arXiv:2101.07577*, 2021.

[84] Weiyang Liu, Yandong Wen, Zhiding Yu, and Meng Yang. Large-margin softmax loss for convolutional neural networks. In *ICML*, volume 2, page 7, 2016.

[85] Yiding Liu, Tuan-Anh Nguyen Pham, Gao Cong, and Quan Yuan. An experimental evaluation of point-of-interest recommendation in location-based social networks. *Proceedings of the VLDB Endowment*, 10(10):1010–1021, 2017.

[86] Pauline Luc, Camille Couprie, Soumith Chintala, and Jakob Verbeek. Semantic segmentation using adversarial networks. *arXiv preprint arXiv:1611.08408*, 2016.

[87] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 7827–7838, 2018.

[88] Yuanfei Luo, Mengshuo Wang, Hao Zhou, Quanming Yao, Wei-Wei Tu, Yuqiang Chen, Wenyuan Dai, and Qiang Yang. Autocross: Automatic feature crossing for tabular data in real-world applications. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1936–1945, 2019.

[89] Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, 2015.

[90] Tariq Mahmood and Francesco Ricci. Learning and adaptivity in interactive recommender systems. In *Proceedings of the ninth international conference on Electronic commerce*, pages 75–84. ACM, 2007.

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

[92] Scott Menard. *Applied logistic regression analysis*, volume 106. Sage, 2002.

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

[94] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

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

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

[97] Hanh TH Nguyen, Martin Wistuba, Josif Grabocka, Lucas Rego Drumond, and Lars Schmidt-Thieme. Personalized deep learning for tag recommendation. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 186–197. Springer, 2017.

[98] Alessandro Nuara, Francesco Trovo, Nicola Gatti, and Marcello Restelli. A combinatorial-bandit algorithm for the online joint bid/budget optimization of pay-per-click advertising campaigns. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[99] Peter Pastor, Heiko Hoffmann, Tamim Asfour, and Stefan Schaal. Learning and generalization of motor skills by learning from demonstration. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 763–768. IEEE, 2009.

[100] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*, pages 4095–4104, 2018.

[101] David Martin Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[102] Xuebin Qin, Zichen Zhang, Chenyang Huang, Chao Gao, Masood Dehghan, and Martin Jagersand. Basnet: Boundary-aware salient object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7479–7489, 2019.

[103] Yanru Qu, Han Cai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. Product-based neural networks for user response prediction. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 1149–1154. IEEE, 2016.

[104] Md Atiqur Rahman and Yang Wang. Optimizing intersection-over-union in deep neural networks for image segmentation. In *International symposium on visual computing*, pages 234–244. Springer, 2016.

[105] Logesh Ravi and Subramaniyaswamy Vairavasundaram. A collaborative location based travel recommendation system through enhanced rating prediction for the group of users. *Computational intelligence and neuroscience*, 2016, 2016.

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

[107] Paul Resnick and Hal R Varian. Recommender systems. *Communications of the ACM*, 40(3):56–58, 1997.

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

[109] David Rohde, Stephen Bonner, Travis Dunlop, Flavian Vasile, and Alexandros Karatzoglou. Recogym: A reinforcement learning environment for the problem of product recommendation in online advertising. *arXiv preprint arXiv:1808.00720*, 2018.

[110] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[111] Stéphane Ross, Narek Melik-Barkhudarov, Kumar Shaurya Shankar, Andreas Wendel, Debadeepta Dey, J Andrew Bagnell, and Martial Hebert. Learning monocular reactive uav control in cluttered natural environments. In *2013 IEEE international conference on robotics and automation*, pages 1765–1772. IEEE, 2013.

[112] Michael H Rothkopf. Thirteen reasons why the vickrey-clarke-groves process is not practical. *Operations Research*, 55(2):191–197, 2007.

[113] Konstantin Salomatin, Tie-Yan Liu, and Yiming Yang. A unified optimization framework for auction and guaranteed delivery in online advertising. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2005–2009. ACM, 2012.

[114] Eric M Schwartz, Eric T Bradlow, and Peter S Fader. Customer acquisition via display advertising using multi-armed bandit experiments. *Marketing Science*, 36(4):500–522, 2017.

[115] Suvash Sedhain, Aditya Krishna Menon, Scott Sanner, and Lexing Xie. Autorec: Autoencoders meet collaborative filtering. In *Proceedings of the 24th international conference on World Wide Web*, pages 111–112, 2015.

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

[117] Jing-Cheng Shi, Yang Yu, Qing Da, Shi-Yong Chen, and An-Xiang Zeng. Virtual-taobao: Virtualizing real-world online retail environment for reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4902–4909, 2019.

[118] Hyungseok Song, Hyeryung Jang, Hai Tran Hong, Seeun Yun, Donggyu Yun, Hyoju Chung, and Yung Yi. Solving continual combinatorial selection via deep reinforcement learning. 2019.

[119] Qingquan Song, Dehua Cheng, Hanning Zhou, Jiyan Yang, Yuandong Tian, and Xia Hu. Towards automated neural interaction discovery for click-through rate prediction. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 945–955, 2020.

[120] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. Autoint: Automatic feature interaction learning via self-attentive neural networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 1161–1170, 2019.

[121] Ramakrishnan Srikant, Sugato Basu, Ni Wang, and Daryl Pregibon. User browsing models: relevance versus examination. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 223–232. ACM, 2010.

[122] Peter Sunehag, Richard Evans, Gabriel Dulac-Arnold, Yori Zwols, Daniel Visentin, and Ben Coppin. Deep reinforcement learning with attention for slate markov decision

processes with high-dimensional states and actions. *arXiv preprint arXiv:1512.01124*, 2015.

[123] Jaeyong Sung, Seok Hyun Jin, and Ashutosh Saxena. Robobarista: Object part based transfer of manipulation trajectories from crowd-sourcing in 3d pointclouds. In *Robotics Research*, pages 701–720. Springer, 2018.

[124] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

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

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

[127] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[128] Yong Kiam Tan, Xinxing Xu, and Yong Liu. Improved recurrent neural networks for session-based recommendations. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, pages 17–22, 2016.

[129] Liang Tang, Romer Rosales, Ajit Singh, and Deepak Agarwal. Automatic ad format selection via contextual bandits. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1587–1594. ACM, 2013.

[130] Michael Tsang, Dehua Cheng, Hanpeng Liu, Xue Feng, Eric Zhou, and Yan Liu. Feature interaction interpretability: A case for explaining ad-recommendation systems via neural interaction detection. *arXiv preprint arXiv:2006.10966*, 2020.

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

[132] Jun Wang, Lantao Yu, Weinan Zhang, Yu Gong, Yinghui Xu, Benyou Wang, Peng Zhang, and Dell Zhang. Irgan: A minimax game for unifying generative and discriminative information retrieval models. In *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval*, pages 515–524. ACM, 2017.

[133] Suhang Wang, Yilin Wang, Jiliang Tang, Kai Shu, Suhas Ranganath, and Huan Liu. What your images reveal: Exploiting visual contents for point-of-interest recommendation. In *Proceedings of the 26th International Conference on World Wide Web*, pages 391–400, 2017.

[134] Weixun Wang, Junqi Jin, Jianye Hao, Chunjie Chen, Chuan Yu, Weinan Zhang, Jun Wang, Yixi Wang, Han Li, Jian Xu, et al. Learning to advertise with adaptive exposure via constrained two-level reinforcement learning. *arXiv preprint arXiv:1809.03149*, 2018.

[135] Xiaobo Wang, Shuo Wang, Cheng Chi, Shifeng Zhang, and Tao Mei. Loss function search for face recognition. In *International Conference on Machine Learning*, pages 10029–10038. PMLR, 2020.

[136] Xiting Wang, Yiru Chen, Jie Yang, Le Wu, Zhengtao Wu, and Xing Xie. A reinforcement learning framework for explainable recommendation. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 587–596. IEEE, 2018.

[137] Yanan Wang, Tong Xu, Xin Niu, Chang Tan, Enhong Chen, and Hui Xiong. Stmarl: A spatio-temporal multi-agent reinforcement learning approach for cooperative traffic light control. *IEEE Transactions on Mobile Computing*, 2020.

[138] Ziyu Wang, Nando De Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. 2015.

[139] Chao-Yuan Wu, Amr Ahmed, Alex Beutel, Alexander J Smola, and How Jing. Recurrent recommender networks. In *Proceedings of the tenth ACM international conference on web search and data mining*, pages 495–503, 2017.

[140] Di Wu, Cheng Chen, Xun Yang, Xiujun Chen, Qing Tan, Jian Xu, and Kun Gai. A multi-agent reinforcement learning method for impression allocation in online display advertising. *arXiv preprint arXiv:1809.03152*, 2018.

[141] Di Wu, Xiujun Chen, Xun Yang, Hao Wang, Qing Tan, Xiaoxun Zhang, Jian Xu, and Kun Gai. Budget constrained bidding by model-free reinforcement learning in display advertising. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 1443–1451. ACM, 2018.

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

[143] Zifeng Wu, Chunhua Shen, and Anton van den Hengel. Bridging category-level and instance-level semantic image segmentation. *arXiv preprint arXiv:1605.06885*, 2016.

[144] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018.

[145] Yuwen Xiong, Renjie Liao, Hengshuang Zhao, Rui Hu, Min Bai, Ersin Yumer, and Raquel Urtasun. Upsnet: A unified panoptic segmentation network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8818–8826, 2019.

[146] Haowen Xu, Hao Zhang, Zhiting Hu, Xiaodan Liang, Ruslan Salakhutdinov, and Eric Xing. Autoloss: Learning discrete schedules for alternate optimization. *arXiv preprint arXiv:1810.02442*, 2018.

[147] Li Xu, Jimmy SJ Ren, Ce Liu, and Jiaya Jia. Deep convolutional neural network for image deconvolution. In *Advances in Neural Information Processing Systems*, pages 1790–1798, 2014.

[148] Min Xu, Tao Qin, and Tie-Yan Liu. Estimation bias in multi-armed bandit algorithms for search advertising. In *Advances in Neural Information Processing Systems*, pages 2400–2408, 2013.

[149] Niannan Xue, Bin Liu, Huifeng Guo, Ruiming Tang, Fengwei Zhou, Stefanos P Zafeiriou, Yuzhou Zhang, Jun Wang, and Zhenguo Li. Autohash: Learning higher-order feature interactions for deep ctr prediction. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[150] Longqi Yang, Yin Cui, Yuan Xuan, Chenyang Wang, Serge Belongie, and Deborah Estrin. Unbiased offline recommender evaluation for missing-not-at-random implicit feedback. In *Proceedings of the 12th ACM Conference on Recommender Systems*, pages 279–287. ACM, 2018.

[151] Scott WH Young. Improving library user experience with a/b testing: Principles and process. *Weave: Journal of Library User Experience*, 1(1), 2014.

[152] Shuai Yuan, Jun Wang, and Maurice van der Meer. Adaptive keywords extraction with contextual bandits for advertising on parked domains. *arXiv preprint arXiv:1307.3573*, 2013.

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

[154] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)*, 52(1):1–38, 2019.

[155] Weinan Zhang, Xiangyu Zhao, Li Zhao, Dawei Yin, Grace Hui Yang, and Alex Beutel. Deep reinforcement learning for information retrieval: Fundamentals and advances. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2468–2471, 2020.

[156] Xiangyu Zhao, Changsheng Gu, Haoshenglun Zhang, Xiwang Yang, Xiaobing Liu, Hui Liu, and Jiliang Tang. Dear: Deep reinforcement learning for online advertising impression in recommender systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 750–758, 2021.

[157] Xiangyu Zhao, Haochen Liu, Hui Liu, Jiliang Tang, Weiwei Guo, Jun Shi, Sida Wang, Huiji Gao, and Bo Long. Autodim: Field-aware embedding dimension searchin recommender systems. In *Proceedings of the Web Conference 2021*, pages 3015–3022, 2021.

[158] Xiangyu Zhao, Chong Wang, Ming Chen, Xudong Zheng, Xiaobing Liu, and Jiliang Tang. Autoemb: Automated embedding dimensionality search in streaming recommendations. *arXiv preprint arXiv:2002.11252*, 2020.

[159] Xiangyu Zhao, Long Xia, Jiliang Tang, and Dawei Yin. Deep reinforcement learning for search, recommendation, and online advertising: a survey. *ACM SIGWEB Newsletter*, (Spring):1–15, 2019.

[160] Xiangyu Zhao, Long Xia, Liang Zhang, Zhuoye Ding, Dawei Yin, and Jiliang Tang. Deep reinforcement learning for page-wise recommendations. In *Proceedings of the 12th ACM Recommender Systems Conference*, pages 95–103. ACM, 2018.

[161] Xiangyu Zhao, Long Xia, Lixin Zou, Hui Liu, Dawei Yin, and Jiliang Tang. Whole-chain recommendations. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 1883–1891, 2020.

[162] Xiangyu Zhao, Long Xia, Lixin Zou, Hui Liu, Dawei Yin, and Jiliang Tang. Usersim: User simulation via supervised generativeadversarial network. In *Proceedings of the Web Conference 2021*, pages 3582–3589, 2021.

[163] Xiangyu Zhao, Tong Xu, Qi Liu, and Hao Guo. Exploring the choice under conflict for social event participation. In *International Conference on Database Systems for Advanced Applications*, pages 396–411. Springer, 2016.

[164] Xiangyu Zhao, Liang Zhang, Zhuoye Ding, Long Xia, Jiliang Tang, and Dawei Yin. Recommendations with negative feedback via pairwise deep reinforcement learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1040–1048. ACM, 2018.

[165] Xiangyu Zhao, Liang Zhang, Zhuoye Ding, Dawei Yin, Yihong Zhao, and Jiliang Tang. Deep reinforcement learning for list-wise recommendations. *arXiv preprint arXiv:1801.00209*, 2017.

[166] Xiangyu Zhao, Xudong Zheng, Xiwang Yang, Xiaobing Liu, and Jiliang Tang. Jointly learning to recommend and advertise. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3319–3327, 2020.

[167] Guanjie Zheng, Fuzheng Zhang, Zihan Zheng, Yang Xiang, Nicholas Jing Yuan, Xing Xie, and Zhenhui Li. Drn: A deep reinforcement learning framework for news recommendation. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 167–176. International World Wide Web Conferences Steering Committee, 2018.

[168] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1059–1068, 2018.

[169] Jieming Zhu, Jinyang Liu, Shuai Yang, Qi Zhang, and Xiuqiang He. Fuxictr: An open benchmark for click-through rate prediction. *arXiv preprint arXiv:2009.05794*, 2020.

[170] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[171] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

[172] Lixin Zou, Long Xia, Zhuoye Ding, Jiaxing Song, Weidong Liu, and Dawei Yin. Reinforcement learning to optimize long-term user engagement in recommender systems. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2810–2818, 2019.

[173] Lixin Zou, Long Xia, Zhuoye Ding, Dawei Yin, Jiaxing Song, and Weidong Liu. Reinforcement learning to diversify top-n recommendation. In *International Conference on Database Systems for Advanced Applications*, pages 104–120. Springer, 2019.

[174] Lixin Zou, Long Xia, Pan Du, Zhuo Zhang, Ting Bai, Weidong Liu, Jian-Yun Nie, and Dawei Yin. Pseudo dyna-q: A reinforcement learning framework for interactive recommendation. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, pages 816–824, 2020.

[175] Lixin Zou, Long Xia, Yulong Gu, Xiangyu Zhao, Weidong Liu, Jimmy Xiangji Huang, and Dawei Yin. Neural interactive collaborative filtering. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 749–758, 2020.