上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

THESIS OF BACHELOR

论文题目：基于深度强化学习的自适应推荐系统

学生姓名：	顾　乡
学生学号：	5130309729
专　　业：	计算机科学与技术专业
指导教师：	张伟楠教授
学院 (系)：	电子信息与电气工程学院

RECOMMENDATION SYSTEM – A MODEL-BASED REINFORCEMENT LEARNING APPROACH

ABSTRACT

Reinforcement learning (RL) has attracted substantial attention over the past several years and has now become a prevalent topic. This paper attempts to apply RL algorithm to recommendation systems when users' feedback is provided online through interaction with the recommendation system. I hope to use RL algorithm to build a system that is able to learn from that online interactive feedback to refine and improve its recommendation policy. In particular, this paper focuses on model-based RL algorithms, a set of algorithms in RL that build a model about the environment and leverage compressed information in the model to help to speed up the learning process. In contrast, model-free solutions refer to those that do not have a model of the environment and learn solely from interactive experiences. I hope to see that the model-based algorithm, when it is applied to recommendation systems, can be more data-efficient than the model-free solution as it is usually the case in RL.

This paper formulates the interactive recommendation problem as a Markov Decision Process (MDP), designs models and proposed two RL algorithms – one model-based and one model-free – to find the optimal recommendation policy for the MDP. Experiments of those two algorithms are then performed on a simulator that is built to simulate the interaction between the recommendation system and the user because online interaction with real users is expensive and risky. Unfortunately, experiment results cannot show assertive improvement in data-efficiency for the model-based solution and more experiments are required for a better understanding of the proposed algorithms as well as the experiment.

KEY WORDS: Recommendation Systems, Reinforcement Learning, Model-based Solution

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1 INTRODUCTION

With the recent prevalence in reinforcement learning (RL), it has attracted research interests to attempting recommendation systems with RL techniques. But why is this even plausible in the first place? If one thinks about a scenario where a recommendation system recommends some item(s) to the user and the user gives feedback (e.g. rating(s)). After receiving the feedback, the recommendation system learns a bit more about the user so that it can refine its recommendation strategy a bit. It then recommends item(s) to the user again and the user provides feedback and repeat. It is clear that this is a decision-making process where the goal of the recommendation system is to learn the best recommendation strategy (policy) for the user so that some performance objective is optimized. At the same, RL approaches intelligence from trial-and-error, meaning that it aims to learn an optimal strategy for action from interaction, so, it is natural to try applying RL techniques on such interactive recommendation systems. There are two key benefits brought by RL: 1). The recommendation system is able to leverage interactive feedback from the user and use it to improve its recommendation strategy simultaneously; 2) RL technique allows the recommendation system to find such a recommendation strategy that cumulative, rather than immediate, feedback (reward) is targeted. Therefore, enormous but all recent efforts have been made in this direction of research[1][2][3][4][5].

However, a concern that has been raised for RL is its sample-inefficiency. That is, RL algorithms are simply too slow in most cases – they require too many interactive experiences before learning a good policy. One possible, among many others[6], is to use model-based RL approach. Basically, the idea of model-based RL is simple: the algorithm learns and constructs a model of the environment at the time of learning a good policy during the interaction. It then utilizes the information, compressed in the model, of the environment to help to accelerate the learning process. In this thesis, I will try to attempt recommendation systems with a model-based RL solution from problem formulation to model design to algorithm selection and finally to experiments. To the best of my knowledge, this project is the first to use model-based RL for recommendation systems. When I began this project, therefore, my guideline is *completeness*, meaning that I will try to stick to the simplest choices and try to avoid any additional complexity (e.g. if I need a Recurrent Neural Network (RNN), I use the standard RNN unit rather than advanced ones like Long Short Term Memory (LSTM) or Gated Recurrent Unit (GRU)).

The rest of this thesis is organized as follows. In chapter 2, I discuss some of the background knowledge as well as related work for RL and recommendation system. In chapter 3, I formally formulate the interactive recommendation process and propose two RL solutions to tackle this problem – a model-free one and a model-based one. Chapter 4 comes the experiment part of the project where I tested these two proposed algorithms on an interactive recommendation simulator built from a publicly available dataset. I also have some discussion about the experiment results and leave some remaining issues about the experiments. In the last chapter (chapter 5), I briefly summarize this thesis project and the conclusions I can draw from the experiment results. I also put three possible things to try next as future work. Finally, I close this chapter with some look-ahead yet pre-mature thoughts inspired during thinking about this project.

Chapter 2 PRELIMINARIES AND RELATED WORK

In this chapter, I will discuss the required background knowledge in the Preliminaries section (section 2.1) as well as related work on RL for recommendation systems in the Related Work section (section 2.2). Before delving into the details of the problem and solutions, I strongly recommend reading, at least skimming, this chapter especially the preliminaries section.

2.1 Preliminaries

2.1.1 Reinforcement Learning Essences

Reinforcement learning (RL) is inspired and originated from studies in psychology[7] where one obvious but often neglected fact is that we, humans, learn by interacting with the environment and adapt our behavior from observed feedback for our previous behavior (i.e., trial-and-error). Modern RL borrows this inspiration and builds intelligent systems that allow computers to learn from trial-and-error. Basically, those systems are designed to be capable of interacting with the environment, either in a simulated or a real world, and learning to figure out the what the environment is like and thus find certain good ways to act within that environment. Formally, the intelligent entity of interest is called the *agent* and the outside world the *environment*. Sutton and Barto[8] have a go-to textbook for beginners that covers the basics theory, influential applications, frontiers of RL as well as its relation to Psychology and Neuroscience.

2.1.2 Markov Decision Process

A Markov Decision Process (MDP) is the fundamental model of RL as which many problems and applications are formulated. It is a model that was beautifully formulated to capture the essentials of the interactive process between agent and environment, upon which rigid mathematical operations and logical reasoning can be performed. An MDP is defined by the following five elements:

- **State Space** $\mathcal{S}$: the set of all possible states. It can either be discrete or continuous.
- **Action Space** $\mathcal{A}$: the set of all possible legal actions in each state. We assume that the possible legal actions in each state is the same.
- **Reward** $\mathcal{R}(\mathcal{S} \times \mathcal{A} \rightarrow [0, 1])$: reward that the agent receives after taking action $\mathcal{A}$ in state $\mathcal{S}$.
- **Transition Probability** $\mathcal{P}(\mathcal{S}'|\mathcal{S}, \mathcal{A})$: the probability of MDP transitioning into state $\mathcal{S}'$ after taking action $\mathcal{A}$ in state $\mathcal{S}$.
- **Discount Factor** $\gamma \in (0, 1]$: a parameter that balances value of delayed rewards as opposed to immediate rewards. 1 means treating all rewards equally whenever it happen; 0, in the other extreme, stands for caring only about the immediate reward.

A classic diagram that captures the interaction is shown in figure 2–1 and the interaction generates a

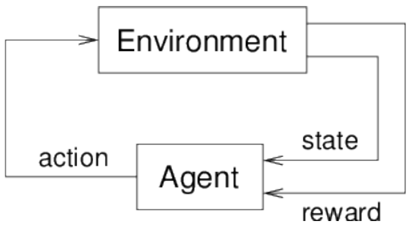


Figure 2–1 A classic diagram of the interaction process between the agent and the environment.

typical data stream, which is also referred to as experiences in this thesis, as

$$S_1, A_1, R_1, S_2, A_2, R_2, \ldots$$

. The quantity that RL tries to maximize is the so-called cumulative future rewards (or, return)

$$G_t = E_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots]$$

, where the subscription π implies that this expectation is over (random) actions $A_t, \forall t$ chosen from policy $\pi : S \times A \rightarrow [0, 1]$. A later proven useful idea is coming up with a concept called state-values. It represents how many cumulative rewards the agent expects to receive if it starts from that state and follows a policy π:

$$v_\pi(s) = E_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots | S_t = s]$$

When the problem naturally breaks into episodes and the data stream of each episode terminates in a finite number of steps, RL algorithms are usually designed to find the optimal policy π^* that maximize state-value of the starting state – the cumulative rewards starting from the initial state (or states if the first state is chosen according to a probabilistic distribution) and follow the policy thereafter.

2.1.3 Solving an MDP – Planning or Learning

How do we actually find that optimal policy then? Before answering that question, it is worth mentioning that the concept, MDP, is not unique to RL. RL is not the first subject that studies it. Some traditional subjects like control theory already have studied it extensively and there exists a systematic solution for MDP and other similar models like Markov Chain, Markov Reward Process with a confusing name – *dynamic programming* (DP). Basically, what it does is assuming perfect information about the environment and use information about the environment to iteratively update the state-values of all states until convergence. The converged state values are then the ones that correspond to the state-values of the optimal policy, so the optimal policy for the MDP can be extracted by a simple one-step look-ahead over all possible actions in each state. Such an approach is often called *planning*, because it plans with known (perfect) information about the outside environment. Or, we say that it plans with a (perfect) model of the environment. Here, a model can be as simple as a black-box algorithm that tells the agent what the next state S_{t+1} and reward R_t will be if we select action A_t in some state S_t.

In most problems, however, we do not know all the information about the environment. In some cases, we might even know nothing about the environment – we need to really go out, explore and try stuff in the environment before getting any useful information about the world. In a sense, we want an algorithm that enables the agents to learn step-by-step in an online manner, accumulating its knowledge about the environment over time. In other words, we want an online algorithm that learns the optimal policy over time. Well, the conventional DP is an iterative algorithm already, can we modify it so that it is suitable for the RL setting? Yes, we can! An insightful observation is that each transition tuple (S_t, A_t, R_t, S_{t+1}) can be used exactly in the same manner as in the DP formula, although I didn't specify the formula here. It turns out that the update here, on average, achieves the same effect as that in DP. We have an algorithm that performs online updating from real interactive experience which allows the agent to learn an optimal policy while interacting with the environment. This process is hence usually referred to as *learning* as opposed to planning, which suggests just sitting there and thinking hard.

2.1.4 Model-based RL: Integrate Learning and Planning

One might be wondering about can we just somehow construct a model of the environment, albeit it might be imperfect, and solve the MDP by resorting to planning with this learn model. Yeah, I know my model of the environment is not perfect but I still hope we can learn a good, if not optimal, policy out of it. Yes, that is something totally legitimate to do and it is often referred to as the *model-based* method, whereas learning only from raw experiences without any models of the environment is called *model-free* method. Both methods have their own pros and cons. For example, model-based methods are typically more data-efficient, meaning they are able to learn a good policy from less experience, because one can query the model to generate hypothetical experience which can be used in the same way as real experience to update the policy. But there are headache problems for it as well, such as how to construct a model from experience, what to do when the model is wrong, etc. On the other hand, model-free methods do not have those issues. It is conceptually and implementationally simple but usually requires a lot of experience to learn a good policy, which is undesirable in cases that interactive experiences are expensive to get.

One elegant framework to integrate the learning and planning process is the Dyna architecture[9]. An perfect illustration is shown in figure 2–2. In a nutshell, what it does is 1). learn a model of the environment from real experiences; 2). update the policy with the real experience in a model-free way; 3). generate imaginary experience by querying the model and update the policy with those imaginary experiences as if they were real.

2.1.5 Interactive Recommendation Process

Recommendation system has evolved from its earliest stage through extensive research and has become one of the most important commercial tools for almost all companies with online stores. Adomavicius and Tuzhilin provide an excellent survey on the classic way of doing recommendation systems[10]. From a high level, those recommendation system algorithms fall into three categories: content-based, collaborative filtering, and hybrid. Content-based recommendation systems recommend items similar to the ones the user liked in the past. Collaborative filtering recommendation systems recommend items that other users with similar taste liked in the past. Hybrid recommendation systems combined those two techniques.

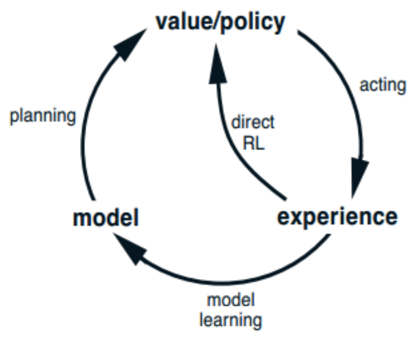


Figure 2–2 The Dyna architecture.

Interactive recommendation systems, however, seek to learn a recommendation policy not only based on offline historical data but also real online feedback from customers. It hopes to build recommendation systems with the ability to provide high-quality recommendation predictions and being able to refine its recommendation policy from up-to-date feedback from the user. This idea coincides with the trial-and-error principle in RL and thus many efforts have gone through adopting RL algorithm to build recommendation systems. Zhao et al. give a concise survey on the current status of using RL for online information seeking, including recommendation systems[11].

When we consider the interactive recommendation process in the context of RL, the agent becomes the recommendation system and the customer to whom the agent provides recommendations to is the environment. It is helpful to think of us being the recommendation system and the customer we are dealing with as the environment. Our goal is then to trial-and-error by trying different recommendations to him/her and gradually figure out his/her preference and taste. It is possible that users' preference might drift over time, but that is okay. RL algorithms have the capability to capture that from the interactive experience stream and learn accordingly for the latest preference.

2.2 Related Work

The related work section here focuses on several key tasks of using RL for recommendation systems.

2.2.1 Exploration v.s. Exploitation

In RL, one long-lasting difficulty is the exploration and exploitation dilemma. It means that, in RL, the agent deploys a policy that it tries to optimize for the maximal cumulative reward to interact with the environment, in order to receive experience for the optimization process. As the agent learns and improves the policy, it will choose those action(s), which have been tried before, with higher reward more often or even always those actions (exploitation). This is fine if we know, from an Oracle point of view, that those actions are indeed the optimal actions but if the agent settles to sub-optimal actions before trying out all possible actions out there (exploration), it might not be able to learn the optimal actions. Exploration, however, often means taking mediocre actions but we have to try it before we know it is not good. What if the agent discovers something really good – actions with higher reward than current best ones? Hence, here is the dilemma we need to face – we want the agent to exploit what it has learned so far to maximize utility/reward while keep exploring for potential better actions.

This dilemma becomes most obvious when the interactive recommendation process is modeled as a contextual bandit problem, where each user corresponds to a context and each item an action. This model concerns the problem of learning the predicted feedback for each action and recommend the ones with the highest predicted feedback. Different users are distinguished by different contexts. The exploration and exploitation dilemma comes where exploitation means to recommend items that are predicted to best match user's preference and exploration means to recommend items randomly to collect more users' feedback. The typical solution for this dilemma is the use of the ϵ-greedy[12], EXP3[13], and UCB1[14].

2.2.2 Dynamic Feedback Environment – Varying Preference

While the performance of most conventional recommendation systems suffer when the users' feedback/preference change over time, RL-based approach is, in its nature, able to cope with this problem. Prior works start studying the possibility of using RL for recommendation with the multi-armed bandit model like we discussed in 2.2.1. Under this setting, people usually introduce an additional variable reward to delineate the dynamic change of the reward function (i.e., the varying users' preference)[15]. Contextual bandit is also presented for such task and several mechanisms, including using estimate confidence for each recommendation action, are proposed to alleviate the dynamic reward change problem[16][17].

Another alternative, maybe a more general solution, is to model the interactive recommendation process as an MDP[2]. This is conceptually feasible because either the multi-armed bandit model or contextual (multi-armed) bandit model are simplifications of the MDP where each episode of the interaction terminates after just one step (that is why the bandit problem is also called a one-step MDP). A state in the MDP corresponds to the user's preference and state transitions represent the preference change in the user. Following this line of work, there are many papers that address different issues of such an approach including large action space[18], page-wise recommendation[3][4], incorporating different scenario information to make recommendation decisions[5], etc.

2.2.3 Maximizing Cumulative Feedback – Engagement in the Long Run

RL is born with the advantage of being capable to find policies that maximize long term cumulative rewards. Does it carry through to the recommendation setting and how does it make sense? Yes, it will be weird if not. If the recommendation system just behaves myopically and recommends the most rewarding item, it is clearly not a good idea if the user always receives the same recommendation. In fact, as pointed out in[19][20], user engagement especially long term user engagement is the essentials of measuring the performance of a recommendation system. Repetitive purchases and recommending a variety of complementary items are an instance of the recommendation system's ability for long term user engagement. Among several related works in this direction, Wu et al.[21] directly models the user's clicks and return behavior in a recommendation system and a bandit-based solution is used to balanced several competing factors including exploitation on immediate clicks, exploitation on expected future clicks and explorations on unknowns that determines the long term user engagement.

Chapter 3 MODEL-BASED REINFORCEMENT LEARNING FOR RECOMMENDATION SYSTEM

This chapter begins the main body of the thesis work. I will, finally, be able to become rigorous in definitions and formula rather than just touching surface as in previous chapters, which meant to serve as a high-level description of the thesis and references related to this work.

3.1 Problem Statement – Formulating Interactive Recommendation Problem as an MDP

Since we want to attempt the recommendation system problem with reinforcement learning, it is a must that we first formulate the interactive recommendation process as an MDP. The recommendation process MDP, serialized by the time step t from 1 to ∞, is defined as follows:

- **State**: A state S_t is defined as the preference of the user, which is encoded from the historical records between the user and the recommendation system $S_t = Encode_\theta(A_1, R_1, \ldots, A_{t-1}, R_{t-1})$, where θ represents the parameters of the encoding network.
- **Action**: An action A_t is defined as one possible item for recommendation.
- **Reward**: The reward R_t is defined as the user's feedback/rating to the latest recommended item A_t under preference S_t.
- **Transition Probability**: The transition is deterministic since states are encoded from historical record, so the next state $S_{t+1} = Encode_\theta(A_1, R_1, \ldots, A_{t-1}, R_{t-1}, A_t, R_t)$.
- **Discount Factor**: a constant $\gamma \in (0, 1]$ that controls the how much we care about future rewards compared to immediate rewards.

The definitions might look rather puzzling if you take your first glance at it. For example, why does Xiang claim the states means preferences? Is it something visible to the agent (recommendation system)? Good question and there are indeed some difficulties in fitting the interactive recommendation process into the MDP framework. I will discuss a few that I can think of in the following sections to try to address those doubts regarding the formulation.

3.1.1 MDP or POMDP?

Typical RL problems, when they are modeled as MDPs, have a clear and straightforward definition for states. For example, in the Acrobot problem[22], the state is defined as the positions and velocities of the bars. Such a definition is reasonable because those pieces of information, on their own, encapsulate all the historical information of the agent, and the future movement (i.e. future positions and velocities) depends only on the current state and the action to be taken. However, in the interactive recommendation process at time step t, all we have is the up-to-date interactive records with the user $A_1, R_1, \ldots, A_{t-1}, R_{t-1}$. How do we define the state then? Well, we do not need to be clear or tricky, we can just use the entire history as our state. Since the interactive records become longer in size as time steps increase, we need to pass it into a network that takes variable length input and outputs a fixed length vector. This vector is then the state representation of the user in this interactive recommendation process (see chapter 3.2.1 for more detail). Although the true state (user's preference) is invisible to us and we have only observations of the action-reward sequence, and the problem should be Partial Observable MDP (POMDP) strictly speaking. We will nevertheless just treat the encoded representations, which is supposed to be a good approximation of the true underlying states, as our states and formulate the whole process as an MDP.

To sum up, we formulate the interactive recommendation process as an MDP because

- It is logically justified to use the whole history of the interactive recommendation process as the definition of states. The encoding network simply transforms the history into a uniformed length representation.
- I simply just don't want to touch POMDP when I have this workaround solution which can save me many headaches caused by nuances and caveats introduced by the partial observation property.
- It turns out, as shown in the experiments section, that this definition actually works fine so it empirically justifies modeling the problem as an MDP.

3.1.2 Episodic or Continuing?

Another natural question to ask is whether this (interactive recommendation) problem is an episodic problem or a continuing one? Well, in principle, when a user can interact with the recommendation system forever, though it probably won't in practice, this interactive recommendation problem is a continuing problem without any designated terminal state(s).

However, if we restrict the maximal number of steps allowed for the interactive recommendation process, say T time steps, it is then a truncated continuing problem with a clear termination signal. In this thesis, I will adopt this definition and treat the MDP as episodic. In practice, each interaction session between the user and the recommendation system won't be too long, so, as long as T is reasonably large, such a conversion is justified.

3.1.3 Continuous or Discrete State/Action Space? Function Approximation or Tabular Solution?

The action space is a (large) discrete space. As per the definition of actions, at each time step t, we can recommend any one of the items, indexed from 1 to the total number of items num_items, to the user $A_t \in \{1, 2, \ldots, num\_items\}$. In fact, we will have one embedding layer for the actions A_t that encodes an integer to a embedding vector (see section 3.2.1 for greater details). Also, I will use actions and items interchangeably in the rest of the thesis according to which word better suits the context.

Now that I decide to use a network to encode states from the historical records $A_1, R_1, \ldots, A_{t-1}, R_{t-1}$, the state space is all possible outcomes in a $\mathcal{R}^d$ space, where d is a hyperparameter chosen from a hyperparameter search. Therefore, the state space is a continuous one.

It is then easy to answer the last question. Yes, we are going to function approximation both for the states and the actions. The embedding layer for actions is a way to extract features for the actions and the encoding layer for states is to extract a good representation of the states. I probably won't be able to interpret what each component in the action/state feature means but they are learned/optimized in a way that best accomplishes the task, no matter what the task exactly is (I will tell you in section 3.2.3).

3.1.4 Recommend One Item or Multiple Items at a Time?

We consider the problem of recommending *one item* at a time, although there are papers[23][3] that specifically studies multiple item recommendation (e.g. page-wise recommendation) as a research direction.

3.1.5 Homogeneous or Heterogeneous Item Repository?

We consider the problem of recommending items in a homogeneous item repository, meaning that all the items we can choose from and recommend are of the same type. For instance, in a movie recommendation scenario, we consider the problem of recommending a movie from a set of movies in a sequence of steps in hopes of maximizing the user's cumulative ratings towards this sequence of recommended movies. If the item repository is heterogeneous, which is closer to an online shopping scenario, one needs to take into account the browsing history of the user since what the user is looking at currently should guide and direct the recommendation choice.

3.1.6 One User or All Users?

Recommendation systems are supposed to build with personalized recommendations. Ideally, we formulate the MDP and interact with a certain user and learn from this user about his/her preference. A learned fair recommendation system is then supposed to recommend items that the user enjoy, from which the cumulative feedbacks are maximized. When this recommendation system serves, each individual user will have a copy of the "base" recommendation system that is able to present reasonably good recommendations and will evolve over time – improving its recommendation performance with the interactive data between the user and the system, so, the longer a user uses the recommendation system (and provides feedback), the better the recommendations will be.

But how do we train such a "base" recommendation system in the first place? We train it via interaction and RL algorithm! A harsh realistic limitation, though, is that it is impossible for a user to interact with the recommendation system to provide sufficient experience for the recommendation system to learn a good policy before the user exhausts. This means that we can have only a very limited number of experiences from a user. What about simulators that can generate simulated interactive experience, one might wonder? Well, it is also hard to build one that perfectly simulates any real person's reaction towards recommended items. We can and we actually will, in the experiment chapter (chapter 4), build such an environment simulator from a publicly available dataset, but, even so, the number of feedback to distinct items from each user is still not adequate. Thus, we cannot expect the recommendation system to learn any useful from limited experience from just one user, although this is a fairly straightforward way to do. Instead, as an alternative solution, we will use experience from all users equally to feed into our RL algorithm and train the model. In this

$$E = k \underbrace{\left[\begin{array}{cccc} \mathbf{0} & \mathbf{1} & \mathbf{\dots} & \mathbf{num\_items} \\ -0.5 & 0.1 & \dots & 2.9 \\ 3.2 & -2.1 & \dots & 3.71 \\ \vdots & \vdots & \vdots & \vdots \\ 7.49 & -4.4 & \dots & -1.1 \end{array}\right]}_{\text{num\_items + 1}}$$

Figure 3–1 An $k \times num\_items$ examplary embedding matrix for actions.

case, our model will still be able to learn something useful to distinguish different since I believe the user information is encoded in the historical records $A_1, R_1, \dots, A_{t-1}, R_{t-1}$ and it becomes a unique identification of the user. Admittedly, when the historical records are empty or short, the model will not be able to know any user information and make specialized recommendations but rather recommend items that are popular to the majority of users. When the historical records get long, the probability of having the same history from two users decreases and the historical records begin to be helpful to distinguish one user from another.

3.2 Design Models and Choose Algorithms to Solve the MDP

In this section, we will formally discuss the models and RL algorithm I designed and chose to build and train a recommendation system. This section is the core part of this thesis project. I will also cover the contribution and novelty of this project – model-based RL approach for recommendation system – in section 3.2.4 and 3.2.5.

3.2.1 State Representation Model

First of all, I used a standard Recurrent Neural Network (RNN) network[24] to build up the state representation model. As stated before, the state representation model functions as a feature extractor that takes as input the historical records $A_1, R_1, \dots, A_{t-1}, R_{t-1}$ at time step t (input length varies with t) and output a fixed-length (d-dimensional) vector S_t.

Before sending the action-reward pairs to the RNN, I used one embedding layer for the actions $A_1, A_2, \dots, A_{t-1}$ to embed each action (an integer indexed from 1 to num_items) to a k-dimensional vector in the embedding space. An examplary embedding matrix $E_{k \times num\_items}$ is shown in figure 3–1. Thus, we can retrieve the embedding for each action using the following operation:

$$E(A_i) = EO_{A_i}, i = \{0, 1, \dots, num\_items\},$$

where O_{A_i} is a $(num\_items + 1)$-dimensional vector with a 1 in A_i^{th} place and 0's everywhere else.

Similarly, I encoded the rewards $R_1, R_2, \dots, R_{t-1}$ via a simple one-hot encoding schema [1] without

[1]This encoding idea is borrowed from Chen et al.[18].

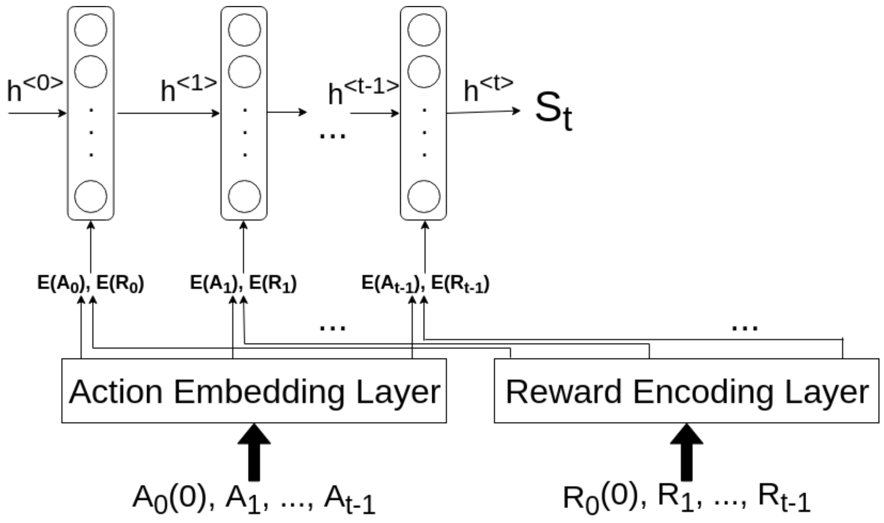


Figure 3–2 The state representation network.

trainable weights:

$$reward\_encoding(R) = onehot(l - floor(\frac{l * (b - r)}{b - a}), l),$$

where $(a, b]$ is the range of the reward R, l the desired dimension after encoding, $floor(x)$ the largest integer no greater than x, and $onehot(i, l)$ an l-dimensional vector with a 1 in the i^{th} position and 0's everywhere else.

The RNN computes the outputs in a looped structure over the (encoded) input action-reward pairs [1] . Formally,

$$\begin{aligned} h^{<1>} &= \tanh(W[h^{<0>}; E_{A_0}; E_{R_0}] + b) \\ h^{<2>} &= \tanh(W[h^{<1>}; E_{A_1}; E_{R_1}] + b) \\ &\vdots \\ S_t = h^{<t>} &= \tanh(W[h^{<t-1>}; E_{A_{t-1}}; E_{R_{t-1}}] + b) \end{aligned}$$

A figure of the state representation model for illustration is shown in figure 3–2. In the rest of the thesis, all models start with the output of the state representation network and I will just refer to the input of those models as state S_t although, in reality, the input should be the historical records $A_1, R_1, \ldots, A_{t-1}, R_{t-1}$ and S_t is computed through a forward pass of the state representation network.

3.2.2 Policy Model

The (recommendation) policy model is responsible for selecting actions/items to recommend at certain state S_t (well the historical records actually). The historical records $A_1, R_1, \ldots, A_{t-1}, R_{t-1}$ will be fed into the state representation model, followed by two fully connected layers to compute the probabilities of selecting each action.

[1]We can also use other advanced RNN units like Long Short Term Memory (LSTM)[25], Gated Recurrent Unit (GRU)[26], etc.

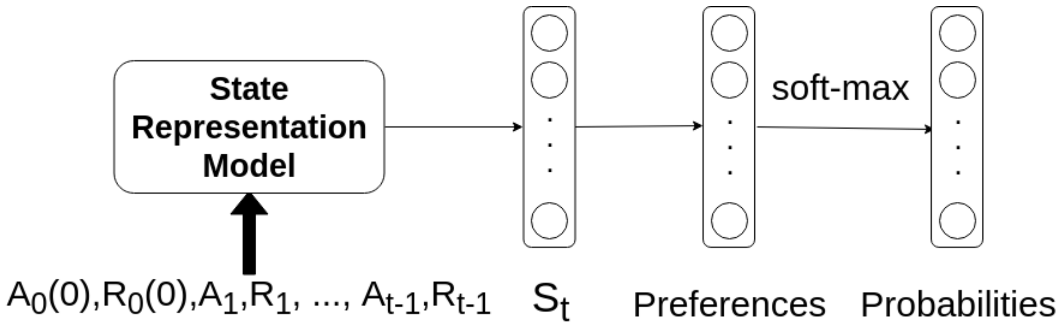


Figure 3–3 The policy network.

The first one layer connects the state representation output (a d dimensional vector) to a layer with num_items nodes with linear combination:

$$h_t = \Theta S_t$$

, where Θ is a $num\_items \times d$ weight matrix [1]. Each component in vector h_t stands for the *preference* of choosing that action in current state S_t.

The second layer is a soft-max layer that normalize those preferences into probabilities P_t so that they sum up to 1. Each component of P_t is computed as follows:

$$P_{t_i} = \frac{e^{h_{t_i}}}{\sum_{j=1}^{num\_items} e^{h_{t_j}}}$$

A figure of the policy model is presented in figure 3–3.

3.2.3 Policy Gradient Method (A Model-Free Solution)

Having set up the necessary models, we can now go and pick one RL algorithm to solve the interactive recommendation MDP. As mentioned in section 2.1.3, we can either do it in a model-free (learning) or model-based (planning) way. In this section, I am going to discuss one model-free solution that I used in the experiments, which will later be incorporated into a model-based solution through the Dyna architecture. (Don't worry if you do not what it means at this moment.)

3.2.3.1 Value-based Solution v.s. Policy-based Solution

In an orthogonal dimensional to the model-free and model-based solutions, there are two categories of solutions for MDPs in RL – value-based and policy-based solution.

Value-based solutions use state-values $v_\pi(s)$ or action-values $q_\pi(s, a)$ as an intermediate quantity to find the optimal policy. Recall from section 2.1.2 that RL rolls around with this central data sequence

$$S_1, A_1, R_1, S_2, A_2, R_2, \ldots$$

[1]Note that I did not use any non-linear activation function here, so the preferences are a linear function of the state representation

State-values are then defined as the expected cumulative rewards if the agent starts from some state and follows some policy π afterwards

$$v_\pi(s) = E_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots |S_t = s]$$

. Similarly, action-values are defined as the expected cumulative rewards if the agent starts from some state and take certain action and follow some policy π thereafter

$$q_\pi(s, a) = E_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots |S_t = s, A_t = a]^1$$

. Thus, a very basic and fundmental operation in RL is to compute the state-values (or actions-values) for a given policy π, which is also often called *policy evaluation* or *prediction*. In a general setting without any information about the environment and learning has to reply on raw interactive experience, prediction uses the following equation to update its state-values (or action-values):

$$V(S_t) \leftarrow V(S_t) + \alpha(R_t + V(S_{t+1}) - V(S_t))$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_t + Q(S_{t+1}) - Q(S_t, A_t))$$

, where $V(s)$ and $Q(s, a)$ are our estimate of the true state-value and action-value respectively and each state (state-action pair) has one such estimate entry. In other words, we keep a large table of estimates of states (state-action pairs) and update them with real-time interactive experience. So long as each state (state-action pair) is visited infinitely many times, those estimates will converge to the true state-values (action-values). For problems with huge state space or even continuous state space (e.g.[27]) that we can no long store an entry for each state (state-action pair), function approximation (FA) is required. Basically, we extract a feature vector for each state (state-action pair) and maintain a function approximator $\hat{v}(s)$ ($\hat{q}(s, a)$) parameterized by w, either a linear one or a non-linear one of the feature vector. We then update the parameters w of the function approximator with the following equation:

$$w \leftarrow w + (R_t + \hat{v}_w(S_{t+1}) - \hat{v}_w(S_t))\nabla_w \hat{v}_w(S_t)$$

$$w \leftarrow w + (R_t + \hat{q}_w(S_{t+1}, A_{t+1}) - \hat{q}_w(S_t, A_t))\nabla_w \hat{q}(S_t, A_t)$$

.

I used an ANN in the thesis project as a state-value approximator and used the update equation above to update the parameter of the state-value prediction network. The detailed usage of this network is delayed in section 3.2.3.2 but now it is enough to understand how to train a neural network as a state-value approximator after receiving one step transition (S_t, A_t, R_t, S_{t+1}) via interaction. This network connects to the output of the state representation network followed by one fully connected layer without any non-linear activation operation, so, I essentially used a linear function approximator to approximate the state-values. An illustration of the network is shown in figure 3–4.

[1] so, the only different between state-values and action-values are that state-values are averages of action-values weighted by the probability of choosing each action under current policy π.

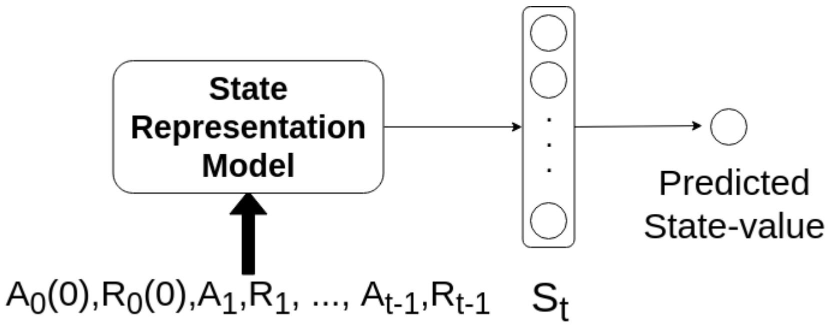

State Representation Model
Predicted State-value
$A_0(0),R_0(0),A_1,R_1, ..., A_{t-1},R_{t-1}$
S_t

Figure 3–4 The state-value prediction network.

Assume we are faced with a problem where a tabular solution is feasible and we have run the previous iterative algorithm for sufficient iterations on the action-values. We end up with a converged action-values $Q(s,a) \approx q_\pi(s,a)$ for some policy π. One thing next we can do is to actually look at the action-values and change the policy from π to some new policy $\pi*$

$$\pi*(s) = argmax_a Q(s,a)$$

and the *policy improvement theorem* ensures that this new policy $\pi*$ is a better policy in the sense that the state-values of $\pi*$ at all states will be larger than or equal to that of π

$$v_{\pi*}(s) \leq v_\pi(s) \forall s \in \mathcal{S}$$

. Now you see why, in processes related to control, action-values are preferred compared with state-values because we can directly extract the most promising action in each state. State-values, on the other hand, need a one-step look-ahead operation to determine the most promising action, which required information of environmental transitions and reward and usually not possible in a learning setting. With function approximation, unfortunately, this policy improvement theorem does not hold anymore but people still use it anyway, and it works fine in most practice.

Policy evaluation (prediction) is important because it provides a way to quantifiably measure the quality or "goodness" of being in each state (or state-action pair) under some policy, and, as you shall see it soon, it actually helps to find the optimal policy as well as its state-values (action-values) by incorporating policy evaluation and policy improvement. The process of finding the optimal policy for an MDP is often called *control* in the literature.

Value-based method refers to solutions that utilize the framework [1] which alternately performs policy evaluation and policy improvement to find the optimal policy as shown in figure 3–5.

Policy-based methods [2], however, abandon the bridging state-values (action-values) and directly manipulate the policy itself to improve its performance. If you think about it, it is actually not surprising at all for such an approach. After all, we want to find the optimal policy to solve the MDP and okay then let's do it –

[1]This framework is also referred to as Generalized Policy Iteration or GPI

[2]Be aware that policy-gradient methods are just one class of policy-based methods that utilize the gradient information. But I will not cover anything about other types of policy-based methods in this thesis.

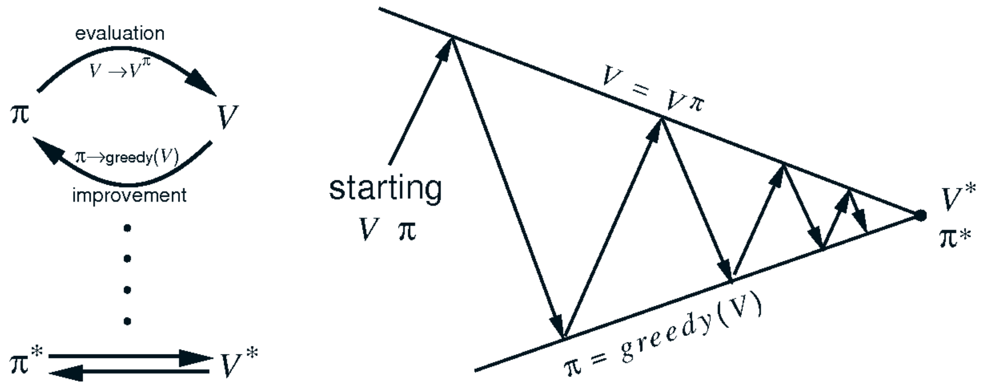


Figure 3–5 The Generalized Policy Iteration framework.

let's try to find a way to improve the policy from some initial random, probably not good either, policy all the way to the optimal policy, hopefully. In fact, the first policy-based solution[28] came before the value-based methods! There are tides going back and forth between those two solutions but recently, policy-based solutions, especially policy-gradient solutions, seem to attract more attention due to its successful application on some hard problems (e.g. robotics control[29]).

Basically, the derivation a policy-gradient method requires the following four procedures:

1. Approximate the policy with parameters θ;
2. Define an objective function that measures the quality of the (parameterized) policy;
3. Compute the derivative of the objective function in terms of online collectible quantities;
4. Interact with the environment to gather data to compute the gradient and use it to update the policy.

It is okay (and you should) if are confused by what those four steps exactly mean. I will discuss more in detail in next section (section 3.2.3.2). But now, allow me to list some of the pros and cons policy-gradient solutions [1]:

Advantages:

- Better convergence properties;
- Effective in high-dimensional or continuous action space;
- Can learn stochastic policies.

Disadvantages:

- Typically converge to a local rather than a global optimum
- Evaluating a policy is typically inefficient and high variance

In this thesis project, I decided to use the policy-gradient method but as you shall see soon, the simple but effective policy-gradient method I picked also uses state-values in it.

[1]Adapted from lectures of Professor David Silver's RL course from UCL

3.2.3.2 One-Step Actor-Critic Algorithm

Now let us walk through the necessary four steps for the policy-gradient method I used (One-step Actor-Critic) and see how we can use the interactive experience to perform gradient ascent to improve our policy.

To begin with, we need to approximate the policy with parameter θ.

$$\pi_\theta(s, a)$$

I already discussed the detail of the network in section 3.2.2. See there for detailed reference.

Secondly, we need to define an objective function that is reasonable for measuring the performance of the policy. Since our interactive recommendation process is treated as an episodic problem (section 3.1.2), a reasonable measurement is the state-value of the initial state [1]

$$J(\theta) = v_{\pi_\theta}(S_1)$$

Thirdly, we need to calculate the derivative of the objective function with respect to θ. Let us first try to derive the gradient of state-values (of any state) w.r.t. θ [2]. To keep the notation simple, it is made implicit in all cases that the gradient is made w.r.t to θ and policy π is parameterized by θ. Also, we assume finite state space but it is true that the derivation generalizes to continuous state space [3]:

$$\begin{aligned}
\nabla v_\pi(s) &= \nabla \left[\sum_a \pi(s, a) q_\pi(s, a) \right] \\
&= \sum_a \left[\nabla \pi(s, a) q_\pi(s, a) + \pi(s, a) \nabla q_\pi(s, a) \right] \\
&= \sum_a \left[\nabla \pi(s, a) q_\pi(s, a) + \pi(s, a) \nabla \sum_{s'} p(s'|s, a) \left(r(s, a) + v_\pi(s') \right) \right] \\
&= \sum_a \left[\nabla \pi(s, a) q_\pi(s, a) + \pi(s, a) \sum_{s'} p(s'|s, a) \nabla v_\pi(s') \right] \\
&= \text{unroll the last term } v_\pi(s') \text{ in the same way} \\
&= \ldots \\
&= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} Pr(s \to x, k, \pi) \sum_a \nabla \pi(s, a) q_\pi(x, a),
\end{aligned}$$

after repeated unrolling, where $Pr(s \to x, k, \pi)$ stands for the probability of transitioning from state s to state

[1] In our problem, the initial state is always encoded from empty historical records so each episode always starts from the same initial state

[2] The derivation is adapted from Chapter 13 of Sutton and Barto's textbook[8]

[3] Essentially, one just needs to replace the summation sign with integral sign for continuous state space.

x in k steps under policy π. It is then indicated that

$$\begin{aligned}
\nabla J(\theta) &= \nabla v_\pi(S_1) \\
&= \sum_s \left(\sum_{k=0}^{\infty} Pr(S_1 \to s, k, \pi) \right) \sum_a \nabla \pi(s,a) q_p i(s,a) \\
&= \sum_s \eta(s) \sum_a \nabla \pi(s,a) q_p i(s,a) \\
&= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a \nabla \pi(s,a) q_p i(s,a) \\
&= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(s,a) q_p i(s,a) \\
&\propto \sum_s \mu(s) \sum_a \nabla \pi(s,a) q_p i(s,a),
\end{aligned}$$

where $\eta(s)$ is the expected number of steps spent on state s in an episode and $\mu(s)$ is the expected fraction of time steps spent on state s in an episode. Great, we know that the gradient of the objective function is proportional to the final term which can be seen as an average of the $\sum_a \nabla \pi_\theta(s,a) q_{pi_\theta}(s,a)$ term weighted by the state visitation frequency. Then if we allow our agent to interact with the environment under policy pi_θ, those states will be encountered in these proportions. Thus, at each time step t, we can compute the $\sum_a \nabla \pi_\theta(S_t,a) q_{pi_\theta}(S_t,a)$ term to obtain a sample gradient whose expectation is equal to the true gradient

$$\begin{aligned}
\nabla J(\theta) &\propto \sum_s \mu(s) \sum_a \nabla \pi(s,a) q_p i(s,a) \\
&= E_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(S_t,a) \right]
\end{aligned}$$

We could have stopped here and instantiate our stochastic gradient ascent algorithm but our current interest is the one-step Actor-Critic algorithm so we can continue our derivation as follows

$$\begin{aligned}
\nabla J(\theta) &= E_\pi \left[\sum_a \pi(S_t,a) q_\pi(S_t,a) \frac{\nabla \pi(S_t,a)}{\pi(S_t,a)} \right] \\
&= E_\pi \left[q_\pi(S_t,A_t) \nabla \ln \pi(S_t,A_t) \right] \\
&= E_\pi \left[G_t \nabla \ln \pi(S_t,A_t) \right] \\
&= E_\pi \left[(G_t - b(S_t)) \nabla \ln \pi(S_t,A_t) \right],
\end{aligned}$$

where $b(S_t)$ is a state-dependent function [1]. This last line holds because the subtracted quantity is zero

$$\begin{aligned}
E_\pi \left[b(S_t) \nabla \ln \pi(S_t,A_t) \right] &= \sum_s \mu(s) \sum_a b(s) \nabla \pi(S_t,a) \\
&= \sum_s \mu(s) b(s) \nabla \sum_a \pi(S_t,a) = \sum_s \mu(s) b(s) \nabla 1 = \sum_s \mu(s) b(s) 0 = 0
\end{aligned}$$

[1]By the way, this is where you begin to see why I said early in the thesis that state-values are also incorporated in effective policy-gradient method.

Usually, people use the state-value as the subtracted state-dependent function (a.k.a baseline) although any function is valid as long as it is state-dependent and does not vary with a [1]. You might be tempted now to turn the expectation into a stochastic gradient descent algorithm to iteratively update the policy parameter θ with sample gradient

$$\theta_{t+1} \leftarrow \theta_t + \alpha \left(G_t - \hat{v}_w(S_t)\right) \nabla_\theta \ln \pi_\theta(S_t, A_t)$$

, along with learning the state-values $\hat{v}_w(s)$

$$w_{t+1} \leftarrow w_t + \beta \left(G_t - \hat{v}_w(S_t)\right) \nabla_w \hat{v}_w(S_t)$$

which is totally fine [2] but still not what we want. One limitation of this algorithm is that we need to wait until the completion of one whole episode to compute $G_t, \forall t \in \{1, 2, \ldots, T\}$ so as to perform the update. How are we going to do about this? Again, use the idea of temporal difference learning – instead of using an unbiased estimate of the state-value G_t, replay it with a biased estimate $R_t + \gamma \hat{v}_w(S_{t+1})$ but of less variance. Formally and fourthly, upon receiving one step of transition (S_t, A_t, R_t, S_{t+1}), we perform the following updates to the policy approximator and state-value approximator

$$\theta_{t+1} \leftarrow \theta_t + \alpha \left(R_t + \gamma \hat{v}_w(S_{t+1}) - \hat{v}_w(S_t)\right) \nabla_\theta \ln \pi_\theta(S_t, A_t)$$

$$w_{t+1} \leftarrow w_t + \beta \left(R_t + \gamma \hat{v}_w(S_{t+1}) - \hat{v}_w(S_t)\right) \nabla_w \hat{v}_w(S_t)$$

. This policy-gradient algorithm is also referred to as the One-step Actor-Critic algorithm. The Actor refers to the policy network that is responsible for taking actions for interaction to gain experience for learning. The Critic refers to the state-value prediction network which tells how good those states are so that it can help to compute the sample gradient and thus improve the Actor. This constitutes the model-free solution for learning a recommendation policy. A complete model-free solution using one-step Actor-Critic for the interactive recommendation process is shown in pseudocode in algorithm 3–1.

3.2.4 Bring in the Local User Model – Reward Predictive Model

As the title of the thesis suggests, the novelty of this thesis project comes from the model-based approach for interactive recommendation process. The first step is to bring in a model of the environment. In general, the model of the environment is something that is able to approximate the transition probability and reward function of the MDP. That is, a model is something that when queried with a state S_t and action A_t, it is capable of returning a reasonable reward R_t and next state S_{t+1} according to the transition probability and reward function. In our problem, however, the state transition is deterministic and can be simply calculated from S_t, A_t, R_t. Thus, the model for the interactive recommendation process turns out to be one for rewards and we hope it is as close to the true reward as possible $\hat{R}_w(s, a) \approx R(s, a)$. Similarly, I used an ANN to approximate the reward function as shown in figure 3–6. It is connected to the state representation network. I concatenate the output S_t with action A_t to a long vector which is then connected to a fully connected layer

[1] An optimal baseline exists[30] but it is rather complicated.

[2] Actually, this is a very classic policy-gradient algorithm called REINFORCE[31]

Algorithm 3–1 A Model-free Solution

```
replay_buffer ← an empty queue of max size $K$
for 1 → $K$ do
    user_id ← randomly select a user id from 1 to $num\_users$
    ob ← an empty list
    for 1 → $T$ do
        ac ← choose an action from the policy network from ob
        rw ← take action ac and receive feedback from this user
        append ac, rw to ob
    end for
    append ob to replay_buffer
end for
train the user model on replay_buffer          ▷ learn a good state representation
loop                                           ▷ start to learn from interaction with users
    user_id ← randomly select a user id from 1 to $num\_users$
    ob ← an empty list
    for 1 → $T$ do
        ac ← choose an action from the policy network from ob
        rw ← take action ac and receive feedback from this user
        invoke One-step Actor-Critic on this transition $(ob, ac, rw)$
        append ac, rw to ob
    end for
end loop
```

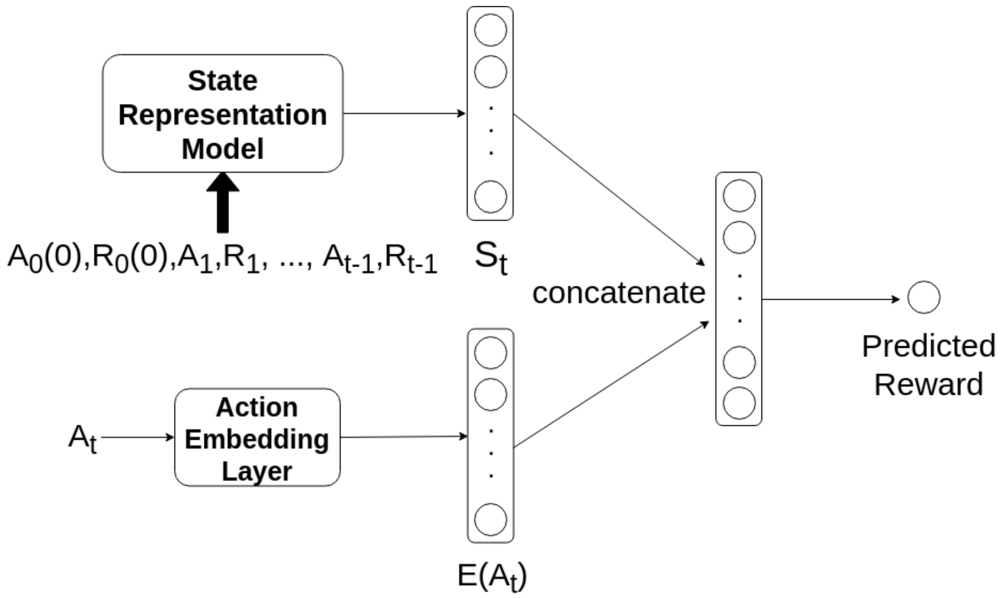


Figure 3–6 The reward prediction network.

to compute the predicted reward for the input state-action pair $\hat{R}_w(S_t, A_t)$. It is then trained, together with the gradient back-propagated to the state representation model [1], with the real transition experience (S_t, A_t, R_t) in a regression manner

$$w \leftarrow w - \alpha \left(\hat{R}_w(S_t, A_t) - R_t\right) \nabla_w \hat{R}_w(S_t, A_t)$$

3.2.5 Integrate Learning and Planning with Policy Gradient Methods via Dyna Framework (A Model-Based Solution)

Suppose we have somehow trained a moderately good user model, we can effectively use it to help accelerate the learning process by integrating planning and learning via the Dyna architecture. As outlined in section 2.1.4, the Dyna architecture learns a model from real interactive experience and uses the model to generate hypothetical experience. It uses either value-based methods or policy-gradient methods to learn to solve the MDP from both real interactive experiences and hypothetical experiences. In this thesis project, I stick to the policy-gradient method (namely, One-step Actor-Critic) as in the model solution in this model-based solution. Basically, all it does is to add a planning module on top of the model-free solution which generates hypothetical (interactive recommendation) transitions with the users and the One-step Actor-Critic algorithm uses it, just as if it is generated from real interaction, to update the recommendation policy network (as well as the state-value prediction network). After each real transition, we will generate several hypothetical experiences since, in practice, simulated experiences are much faster and easier to generate than real interactive experiences.

[1]The state representation and the reward prediction model collectively are also sometimes called the user model.

3.2.5.1 Generate Hypothetical Experiences With Rollouts

One seemingly simple but actually not problem remains: how do we do the planning step? More specifically, which state-action pairs should we query the model to receive the predicted reward and predicted next state so that we can feed this hypothetical transition to our algorithm to improve the policy? It might be not straightforward why this is a problem. Let us give a few scenarios of this problem to help you appreciate the difficulty of this planning process.

Consider a deterministic environment with a tabular solution. That is, we can build a model that stores the reward and next state for each state-action pair $Model(S, A) \rightarrow R, S'$. In this case, we can initialize our model to be empty and as the agent interacts with the environment and receives interactive experience, say, S_t, A_t, R_t, S_{t+1} at time step t, we can just set $Model(S_t, A_t) = R_t, S_{t+1}$. Since the environment is deterministic, meaning that same state-action pair S, A always leads the agent to the same next state S' with same reward R. Then when planning, we can select randomly from *visited* state-actions pairs [1] and query the model because we know the model will give us exactly correct next state and reward. Those hypothetical transitions are thus informative for the agent to learn about the environment.

What if the environment is stochastic? Well, One can still resort to the same idea but now each state-action might lead to different next states and rewards according to some unknown transition probability. We can approximate that probability by storing all the happened next states and rewards with counts, and when query the model, return the next state and reward by sampling from all the possible next states and rewards in accordance to their counts.

In both of the two cases, there is one major constraint: we always select *visited* state-action pairs to query the model. What bother then building a model? Why do not we just store all the experiences and repeatedly use it, rather than discarding it after using it once, to train and improve the policy? This approach sounds like it is doing the same thing as the planning process since we are only going to query visited state-action transitions from the model anyway. Hence, ideally, a good model is one that is not only able to remember visited transitions but also capable of generalizing to new unseen transitions. This property is demanded in particular to problems where function approximation is required. For example, for a problem with continuous state space, we do not want to query the model of exactly the same state-action pair as some previously visited ones. We want to query some unvisited state-action pairs but the model is still able to give us reasonable next state and rewards because, maybe, the state of this new state-action pair is very close to the state in some previously visited state-action pair with the same action, and thus they should have similar next state and reward.

In our interactive recommendation process, the state space is continuous and the action space is discrete but large. I can be "safe" and only query the model on visited transitions to get hypothetical experiences to train the policy. However, as discussed above, this becomes meaningless when we can simply use a replay buffer to store all the experiences and train the policy repeatedly on those experiences in the replay buffer. So, I will still try using a model and I will query the model on unseen state-action pairs to make the

[1]Or, more efficiently, use an strategy that selects those state-action pairs with highest change in their action-values, which often goes under the name of prioritized sweeping[32]

model meaningful in the sense it generates new unseen transitions to train the policy. Namely, the planning procedure, called *rollout*, is conducted like this: At time step t, after the transition (S_t, A_t, R_t, S_{t+1}) with the real environment, the agent starts the planning procedure from state S_{t+1}. It first consults the (current) policy model to select an action A_{t+1} at state S_{t+1}. It then inputs S_{t+1}, A_{t+1} to the reward prediction model to get the predicted reward $\hat{R}_{t+1}$ as well as computing the (hypothetical) next state $\hat{S}_{t+2}$ from $S_{t+1}, A_{t+1}, \hat{R}_{t+1}$ via the state representation network. Now we have one step of hypothetical transition $S_{t+1}, A_{t+1}, \hat{R}_{t+1}, \hat{S}_{t+2}$ and we feed it to the One-step Actor-Critic algorithm to update the (recommendation) policy. The agent then repeats the whole process from $\hat{S}_{t+2}$ for certain allowed planning steps. Then the agent will revert back to state S_{t+1} and pick an action A_{t+1} to interact with the real environment to get true reward R_{t+1} and next state S_{t+1} in time step $t+1$ and update the policy with this latest real transition $S_{t+1}, A_{t+1}, R_{t+1}, S_{t+2}$ before entering the planning procedure again. The name "rollout" refers to the idea of thinking the planning process as continuing on the real experience, so the planning procedure rolls out from a certain point of the real interactive experience.

3.2.5.2 Pre-train and Periodically Re-train The User Model

There is just one more remaining problem before giving out the algorithm for the model-based solution. That is, how do we obtain a reasonable user model in the first place? In the classic Dyna architecture, the model is learned online at interaction together with the actual learning and planning process. In our problem, however, this is not feasible because the planning procedure (rollouts) requires reasonable generalization to new unseen state-action pairs. If the model is trained online, at the early stages of the learning process, the model is immature and the hypothetical experiences can be wildly off. Then it hurts the policy if we train it with those wrong transitions. To address this issue, I pre-trained the user model with some offline historical interactive data. Emmm, does this pre-training process practical in real recommendation context? Yes, in the real recommendation context, there is usually some historical data upon which we can use and train a user model.

To be specific, I used a replay buffer (a fixed-length queue of size K [1]) to store the latest user interactive experiences $(A_1, R_1, \ldots, A_T, R_T)$ where T is the maximal number of time steps for each episode. I first randomly picked K users and interact with each one of them for T steps $(A_1, R_1, \ldots, A_T, R_T)$ to fill the replay buffer with K such episodic experiences. Then, I trained the user model (state representation model and reward prediction model) on the replay buffer by splitting each and every episodic experience into T training samples – $(A_0, R_0, A_1 \to R_1), (A_0, R_0, A_1, R_1, A_2 \to R_2), \ldots, (A_0, R_0, A_1, R_1, \ldots, A_T \to R_T)$ [2]. I used those $K * T$ training samples to pre-train the user model.

Now, I have set up everything and we can start running the model-based solution. There is just one more experimental detail I want to talk about. After interacting with various users in the real environment and gathering new episodic experience, I will append it to the replay buffer [3]. After a certain amount of old episodic experiences being replayed by new episodic experiences, say, three-fourths of the replay buffer size,

[1] Do not confuse it with the lowercase k which refers to the dimension of the action embedding layer!

[2] A_0 and R_0 are both zero. They are used in the place of reward and action inputs when the historical records are empty. In other words, we input (encoding of) zeros to the state representation network to compute the representation of the initial state S_1.

[3] Since the replay buffer is a fixed-length queue, the oldest episodic experience will be replaced by this latest experience.

I will re-train our reward prediction model (but not the state representation model) using experiences in the replay buffer in the same manner as in the pre-training step. Notice that the weights of the state representation model are saved and frozen after the pre-training step for two reasons:

- We will load the weights to the state representation model in the model-free solution so that we can have a fair comparison between the model-based and model-free solution with the same state representation;
- Before periodical re-training, the recommendation system has updated its policy with many interactive transitions and I think the policy "gets used to" that state representation, so updating the state representation model during re-training might hurt the performance of the policy.

Now that I have gone through all the models, algorithms and special procedures, an algorithm of the model-based solution in pseudo code is given in algorithm 3–2.

Algorithm 3–2 A Model-based Solution

```
1:  replay_buffer ← an empty queue of max size $K$
2:  for $1 \to K$ do
3:      user_id ← randomly select a user id from 1 to $num\_users$
4:      ob ← an empty list
5:      for $1 \to T$ do
6:          ac ← choose an action from the policy network from ob
7:          rw ← take action ac and receive feedback from this user
8:          append ac, rw to ob
9:      end for
10:     append ob to replay_buffer
11: end for
12: train the user model on replay_buffer    ▹ learn a good state representation and reward prediction model
13: loop                                     ▹ start to learn from interaction with users as well as planning
14:     repeat
15:         user_id ← randomly select a user id from 1 to $num\_users$
16:         ob ← an empty list
17:         for $1 \to T$ do
18:             ac ← choose an action from the policy network from ob
19:             rw ← take action ac and receive feedback from this user
20:             invoke One-step Actor-Critic on this transition (ob, ac, rw)
21:             append ac, rw to ob
22:             rollout_ob ← a deep copy of ob                  ▹ start planning process
23:             for $1 \to$ max_planning_steps do
24:                 ac ← choose an action from the policy network from rollout_ob
25:                 rw ← take action ac and receive feedback from this user
26:                 invoke One-step Actor-Critic on this transition (rollout_ob, ac, rw)
27:                 append ac, rw to rollout_ob
28:             end for
29:         end for
30:         append ob to replay_buffer    ▹ append with replacement of the oldest element in replay_buffer
31:     until retrain_buffer_ratio of replay_buffer have been replaced
32:     train the user model on replay_buffer            ▹ re-train user model with updated replay_buffer
33: end loop
```

Chapter 4 EXPERIMENTS

Equipped with all the necessary theory and models, I can now move into the experiment part to see how the proposed algorithm works in practice. In particular, the experiments are designed to answer the following two questions:

1. Does the RL-based approach actually work, when the interactive recommendation problem is modeled as an MDP?
2. Is the model-based solution more data efficient compared with the model-free solution as expected?

4.1 Build an Environment Simulator from MovieLens 10M Dataset

Due to the interactive nature of the recommendation problem, it would be ideal to conduct experiments online via interactions with real users but this is often expensive and risky as pointed out in Li's presentation[33]. Instead, I borrowed ideas from prior works cite-large-action-paper-here to build an environment simulator from a publicly available dataset which can simulate the users' feedback (reward) for different recommended items (actions). Since the next state can be computed quite straightforward, this environment simulator is actually a reward simulator. We measure the performance of the proposed algorithm on interactions with this simulator.

Before talking about details of building the simulator, let's take a look at the dataset I used as well as its statistics. I used the MovieLens 10M dataset [1] which contains 10000054 ratings applied to 10677 movies from 69878 users [2]. On average, each user has around 140 ratings, where each rating is between (0,5] with 0.5 increment (e.g. 0.5, 2.5, 5.0). Thus, if we conceptually envision a big rating table where rows are user ids (from 1 to 69878) and columns are movie ids (from 1 to 10677) [3], it will be a very sparse table in the sense that the majority of the entries are unknown. Such a illustration is shown in figure 4–1.

I normalized the ratings from (0,5] to (-1,1] while maintaining the ratio using the following transformation:

$$\text{new\_rating} = 0.4 * \text{old\_rating} - 1^{4}$$

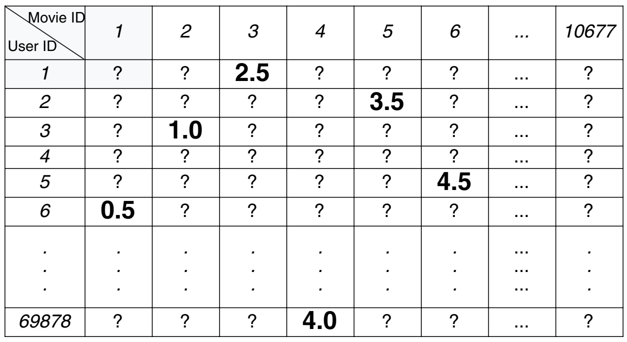
and the environment simulator takes a user id and an item id and return the corresponding entry (after normalization) of the table. If the queried entry is unknown (there is no record for the rating of the user on this specific movie), then we return a 0, implying that we don't know how the user would rate this movie so we guess a neutral rating (2.5, or 0 after normalization). One can imagine that when interacting with

[1] Downloadable from https://grouplens.org/datasets/movielens/10m/

[2] The website http://files.grouplens.org/datasets/movielens/ml-10m-README.html states "...10681 movies by 71567 users" but it is not after careful examination.

[3] In the original dataset, neither user id nor movie id are indexed from 1 to 69878 or 10677 but rather some indexings with gaps, so I re-indexed the user ids and movies ids and the relative ordering might not hold after re-indexing.

[4] We do this simply to re-range the reward to be symmetric around 0. Do not confuse it with the reward embedding layer to which we feed the normalized reward.



Movie ID / User ID	1	2	3	4	5	6	...	10677
1	?	?	**2.5**	?	?	?	...	?
2	?	?	?	?	**3.5**	?	...	?
3	?	**1.0**	?	?	?	?	...	?
4	?	?	?	?	?	?	...	?
5	?	?	?	?	?	**4.5**	...	?
6	**0.5**	?	?	?	?	?	...	?
⋮	⋮	⋮	⋮	⋮	⋮	⋮	...	⋮
69878	?	?	?	**4.0**	?	?	...	?

Figure 4–1 An examplary rating table based on MovieLens 10M dataset.

this simulator to learn a good recommendation policy, the reward for the majority of the actions will be 0, although it is probably not true for online user interaction. The sparse reward property might be an issue for training but we just have to live with that.

4.2 Performance Evaluation Metrics and (Hyper)parameter Choices

I wrote a performance measure procedure that is called to measure the performance of the recommendation policy at that moment during interactions with the environment simulator. Although the workflow of conventional approaches of recommendation systems will split user dataset into a training set and a test set and training stage is separate from test stage, I did not follow this pattern to measure the performance of my proposed algorithms. Instead, I stick to the RL style of measuring RL algorithms which typically records the change of average cumulative rewards (or costs) for one episode as a way to measure the quality of the policy. Since RL algorithms are derived to maximizing cumulative rewards (or minimizing cumulative costs), typically an increasing (or decreasing) learning curve is expected and we are interested in, for example, how fast this curve goes up (or down), the final performance the curve achieves, monotonically changing or not, etc. Following this idea, I simply let the algorithm run and plug in a performance measure procedure after every certain iterations.

Concretely, after every retrain_buffer_ratio*K iterations [1], the measurement procedure is evoked to report the performance of the current policy. The procedure interacts with k users for T time steps each to compute the following four metrics:

- Average Reward: Average reward over all $K * T$ steps;
- Precision@n: At each time step t for user i, look at the n actions with highest probabilities given by current policy on S_t. Then query the environment simulator on user i to see how many items among those n items have an actual rating larger than or equal to some threshold (e.g. 3.0 or 0.2 after normalization); The ratio is then the precision@k for this time step t. Average this quantity over all $K * T$ steps;
- Recall@n: At each time step t for user i, query the environment simulator on user i for all actions

[1] interaction with one user for T time steps counts as one iteration

T (episode length)	32
K (replay buffer size)	3000
d (dimension of state representation)	10
k (dimension of action embedding)	50
l (dimension of reward encoding)	10
n (top-n evaluation metric)	32
γ (discount factor)	0.99
retrain_buffer_ratio	1.0
max_planning_steps	5
a (reward lower bound after normalization)	-1
b (reward upper bound after normalization)	1
num_users (MovieLens 10M specific)	69878
num_items (MovieLens 10M specific)	10677

Table 4–1 The table of (hyper)parameters of my experiment.

with actual ratings larger than or equal to a threshold. Then look at the n actions with the highest probabilities given by current policy on S_t to see how many of those actions show up in those n actions. The ratio is then the recall@n for this time step t. Average this quantity over all $K * T$ steps;

- F1@n: At each time step t for user i, compute f1@n for this time step t using $\frac{2*precision@n*recall@n}{precision@n+precision@n}$. Average this quantity over all $K * T$ steps.

Table 4–1 lists all the (hyper)parameters I used in my experiment in case you want to reproduce my result [1].

4.3 Results and Discussions

The experiment result is shown in figure 4–2. The leftmost column is the result of the model-free solution; The middle column is the result of the model-based solution; The rightmost column simply put the previous two columns into one for comparison. The four rows correspond to the four evaluation metrics – average reward, precision@n, recall@n, and f1@n respectively.

We can see that for the average reward metric, both methods behave similarly. It starts and remains at close to 0 for some time before it (almost) suddenly jumps up and oscillates at around 0.33. The model-based solution has this jump-up before the model-free solution for about 6000 iterations. The model-free solution has a more smooth increase than the model-based solution, but still very rapid. Each method has a similar pattern between the remaining three metrics. All three metrics increase and gradually plateau for the model-free solution The model-based solution, however, encounters a drop in these three metrics in the second half stage of learning, causing it to be eventually lower than the model-free solution.

[1] Or, go to https://github.com/Xiang-Gu/Model_Based_RL_Recommendation_System to see my implementation for reference.

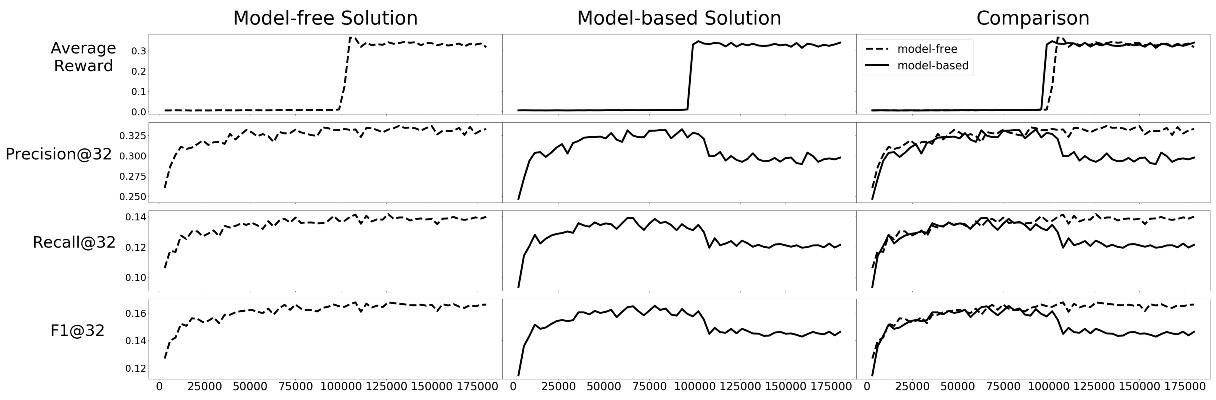


Figure 4–2 The result of my experiment.

4.4 Several Remaining Issues

During the course of this project, I encountered many problems relating to both the theory and the experiment. A project like this is more like an application – an application of RL to recommendation systems – rather than theory, so there are typically many questions with arbitrary choices and I need to make a choice in order to proceed. Many of those choices do not have a definite answer about which is right and which is wrong. All those choices just seem equally good and none of them is better nor worse than others, so it often comes to an answer like this "Okay then, maybe let's do the simplest now, and try out others later if we have time". Unfortunately, due to limited time for this thesis, I did not have the chance to try out all the other possibilities but I have summarized them here and they can be a great guide for next steps of work!

1. I need to understand the behavior of the recommendation policy in greater detail rather than just looking at a figure of the evaluation metrics. For example, what exactly happened to the recommendation policy for that shoot-up in the average reward plots? What are the items it is recommending? Is it just repeatedly recommending the same one or two items? Similarly, what causes the drop in the precision@32, recall@32, and f1@32 for the model-based solution? In the early stage of the learning, why is the precision@32 already very high (around 0.3) when the average reward is close to 0? Furthermore, a case study will be perfect in order to explain the behavior of the recommendation system.
2. Should we update the state representation network when improving the policy with the One-step Actor-Critic algorithm? Recall that all the models (reward prediction model, policy model, state-value prediction model) are connected to the state representation network, so we need to decide when to and when not to update the weights of the state representation model. In my experiment, I update the state representation model only at the pre-training step. Its weights are then frozen so neither Actor-Critic nor periodically re-training will update it. I did this because 1). I want to fix the state representation for a fair comparison between model-free and model-based solution, and 2). I think changing the state representation in the Actor-Critic algorithm or pre-training step might hurt the performance because the policy network, as well as the state-value prediction network, might get

used to that state representation, so changing it might make some of the training efforts go in vain. It will be interesting to see how different strategies of updating the state representation impact the learning process and the final result, and decide then which strategy to use. It is possible that any updating strategy, or even any weights, say the typically worst initial random weights, will actually work similarly without significant difference, then how are we going to explain it?

3. Did I really train the user model in the appropriate way for the model-based solution? Recall that the way I trained the user model is to first fill in a replay buffer of interactive experiences with the environment simulator where each experience is an action-reward sequence of length T $A_1, R_1, \ldots, A_T, R_T$, then construct input-output pairs like $A_0, R_0, A_1 \rightarrow R_1$, $A_0, R_0, A_1, R_1, A_2 \rightarrow R_2$, etc. and finally train our user model on those pairs in a regression manner. However, considering the fact that we build this environment simulator from the MovieLens 10M dataset (a sparse rating table), many of the user-item pair queries will have a simulated reward of 0, because we don't have that rating in the dataset. Then when trained with those rewards, the user model (the reward prediction model in particular) is forced to remember that those actions are neutral (not good nor bad), which might not be the case. I need to find a way to convince myself and others that what the user model predicts – $\hat{R}_t$ given S_t, A_t – is actually a reasonable prediction of the true R_t. For example, I observed that during the pre-train step, the loss (mean squared error) on each of the training sample (I used stochastic gradient descent) is very close to zero and I also observed that during the planning step where the model generates hypothetical transitions, mainly the hypothetical rewards, the majority of those (hypothetical) rewards are also very close to 0. These two shreds of evidence are not enough though – maybe the user model simply learns to output a zero regardless of the input because we have so many zero rewards in the training set. Then the planning step will not have any meaningful transitions because all the predicted rewards will just be some noise around zero. I need to further make sure that for those actions/items with non-zero rating, our user model actually predicts some reward significantly away from zero when one of those items is the selected action for that time step. Hence, probably a better idea is to train the user model only on input-output pairs $A_0, R_0, A_1, R_1, \ldots, A_t \rightarrow R_t$ where R_t is an actual rating in the dataset, rather than unknown (and thus a reward of 0 is guessed).
4. I did not perform hyperparameter tuning yet due to limited time. It will be necessary to do hyperparameter tuning for better experimental results.
5. Are precision@n, recall@n and f1@n really reasonable evaluation metrics for this problem? Those metrics are popular and reasonable evaluation metrics in conventional approaches, but are they in this RL-based approach?
   - for precision@n, what if some user has just a few (say < n) relevant items (i.e. items that have a actual rating larger than or equal to some threshold)? Then precision@n for this user will never be able to get close to 1 no matter how good the recommendation policy is, because the numerator is always less than the denominator.
   - for recall@n, what if some user has NO relevant items (denominator is 0), should we set the

recall@n for this user to 1 (I did in the experiments), as we do in conventional recommendation systems? Also, the denominator is related to number of relevant items of the user, rather than related to n, so the range of recall@n is no longer [0,1] depending on the quality of the recommendation policy. Thus, the recall@n quantity can no longer reflect the goodness of the recommendation policy since it is also related to the user.

- for f1@n, if precision@n and recall@n do not make sense in the first place, how can f1@n, one quantity that relies on precision@n and recall@n, make sense?

Chapter 5 CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, I tried to attempt the recommendation system with an RL-based approach. Although there are several prior works on applying RL algorithms to recommendation system, this thesis focuses on the model-based RL approach, in the hopes of increasing data efficiency.

I formulated the interactive recommendation problem as an MDP and discussed some questions regarding this formulation. Then I designed models and chose RL algorithms to solve this MDP. Finally, I turned all that to code and experimented with a publicly available dataset. The results I get is not enough to give any conclusive statement, although I very much want to say that "yes, the model-based solution is more data efficient than the model-free solution". There are several weirdnesses in the resultant figure and a better understanding of the behavior of the recommendation policy is required. I listed some of the remaining issues as well as some potential solution which can be followed as the next steps to better understand the proposed algorithms.

5.2 Future Work

Obviously, the remaining issues and their potential solutions as discussed in section 4.4 would be the things to try first. In addition, I have three more things that I think it is worth trying as future work about this project:

1. Implement some more baselines for this problem such as the Random policy, the Popularity policy, the greedy-SVD policy.
2. The updating formula for the One-step Actor-Critic in section 3.2.3.2 is only partially correct. Thomas[34] noticed that the update formula as specified in that section is actually a biased estimate of the true policy gradient cite-Thomas-paper-here. An unbiased update should be

$$\theta_{t+1} \leftarrow \theta_t + \alpha\gamma^t \left(R_t + \gamma\hat{v}_w(S_{t+1}) - \hat{v}_w(S_t)\right) \nabla_\theta \ln \pi_\theta(S_t, A_t)$$

   with an addition γ^t term in it. The reason why most people uses the "incorrect" update formula is probably that this is only a recent discovery in 2014 and it seems like things are just working fine with the old update formula. Indeed, the old update formula is not way off from the new one; they differ by just γ^t so I guess people just keep using the old formula since the difference is buried in good results. In my experiment, I also used the old update formula. It will be interesting to see whether adding the γ^t term will make any noticeable changes to the result.
3. Try multi-step methods or eligibility trace for the Actor-Critic algorithm. Currently, I used One-step Actor-Critic algorithm but, in general, using multi-step methods or eligibility trace will greatly boost the learning speed. This is shown in several classic RL problems (see page 318 of Barto and Sutton's textbook[8] as well as chapter 12 for a more detailed discussion about eligibility trace).

4. What about a model-free solution with experience replay? I have spent a lot of time and effort on the user model, trying to find a good way to train and query the model in order to use the model to speed up learning. Why don't I go simple about things and just use the model-free solution but storing all the experience and repeatedly train the policy on those experience? In my current model-based approach, I store all those experience in the replay buffer anyway so this potential alternative does not incur extra memory complexity.

5.3 Some Pre-mature Thoughts: Planning and Acting in Non-observable Domains

When I was thinking and working on this project, one thing that is consistently popping up is the historical records of the recommendation interaction

$$A_1, R_1, \ldots, A_t, R_t$$

Compare it to the essential data stream in RL

$$S_1, A_1, R_1, \ldots, S_t, A_t, R_t$$

You will see that our interaction is exactly the same as this one without states. In this thesis project, I just created a definition for states (from partial historical records) although there is not really a straightforward interpretation of states in the interactive recommendation process. I did this just to fit the problem into the MDP framework so that I can proceed with the RL concepts and algorithm which is built upon the MDP framework. Some paper[11] said, "Under the MDP setting, the state is introduced to represent user' s preference and state transition captures the dynamic nature of user' s preference over time." It seems to me that this state representation is just like an embedding layer that transforms the historical records into a vector in the state space and we cannot really have an interpretation for what each component/dimension of the state representation means.

Another similar framework is the POMDP which is built upon the sequence

$$O_1, A_1, R_1, \ldots, O_t, A_t, R_t$$

where O_t stands for the *observation* at time step t, which the things the agent actually get to see. Loosely speaking, one can think of the observation as partial information of the true underlying state. There are some excellent works[35][36] in this area. I was thinking about what if there is nothing observable for the agent and all we have is the action-reward sequence like in this project. I don't know whether the POMDP theory covers this situation (i.e. $O_t = 0 \forall t$) so I guess I need to learn more about this area. If not, then I think we can always use the idea, as we did in this project, of encoding all the action-rewards up until certain time step t to get an artificial state $S_t = Encode(A_1, R_1, \ldots, A_{t-1}, R_{t-1})$, or there is something more complicated and more effective than this that can be done by, for example, getting inspiration from the POMDP theory.

Bibliography

[1] DULAC-ARNOLD G, EVANS R, van HASSELT H, et al. Deep reinforcement learning in large discrete action spaces[J]. ArXiv preprint arXiv:1512.07679, 2015.

[2] ZHAO X, ZHANG L, DING Z, et al. Deep reinforcement learning for list-wise recommendations[J]. ArXiv preprint arXiv:1801.00209, 2017.

[3] ZHAO X, XIA L, ZHANG L, et al. Deep reinforcement learning for page-wise recommendations[C]// Proceedings of the 12th ACM Conference on Recommender Systems. ACM. [S.l.]: [s.n.], 2018: 95–103.

[4] ZHAO X, ZHANG L, DING Z, et al. Recommendations with negative feedback via pairwise deep reinforcement learning[C]// Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. ACM. [S.l.]: [s.n.], 2018: 1040–1048.

[5] ZHAO X, XIA L, ZHAO Y, et al. Model-Based Reinforcement Learning for Whole-Chain Recommendations[J]. ArXiv preprint arXiv:1902.03987, 2019.

[6] BOTVINICK M, RITTER S, WANG J X, et al. Reinforcement Learning, Fast and Slow[J]. Trends in cognitive sciences, 2019.

[7] KLOPF A H. Brain function and adaptive systems: a heterostatic theory[R]. AIR FORCE CAMBRIDGE RESEARCH LABS HANSCOM AFB MA, 1972.

[8] SUTTON R S, BARTO A G. Reinforcement learning: An introduction[M]. [S.l.]: MIT press, 2018.

[9] SUTTON R S. Dyna, an integrated architecture for learning, planning, and reacting[J]. ACM SIGART Bulletin, 1991, 2(4): 160–163.

[10] ADOMAVICIUS G, TUZHILIN A. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions[J]. IEEE Transactions on Knowledge & Data Engineering, 2005(6): 734–749.

[11] ZHAO X, XIA L, TANG J, et al. Reinforcement Learning for Online Information Seeking[J]. ArXiv preprint arXiv:1812.07127, 2018.

[12] WATKINS C J C H. Learning from Delayed Rewards[D/OL]. Cambridge, UK: King's College, 1989. `http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf`.

[13] AUER P, CESA-BIANCHI N, FISCHER P. Finite-time analysis of the multiarmed bandit problem[J]. Machine learning, 2002, 47(2-3): 235–256.

[14] AUER P, CESA-BIANCHI N, FREUND Y, et al. The nonstochastic multiarmed bandit problem[J]. SIAM journal on computing, 2002, 32(1): 48–77.

[15] ZENG C, WANG Q, MOKHTARI S, et al. Online context-aware recommendation with time varying multi-armed bandit[C]// Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. ACM. [S.l.]: [s.n.], 2016: 2025–2034.

[16] WU Q, IYER N, WANG H. Learning Contextual Bandits in a Non-stationary Environment[C]// The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval. ACM. [S.l.]: [s.n.], 2018: 495–504.

[17] LIU F, LEE J, SHROFF N. A change-detection based framework for piecewise-stationary multi-armed bandit problem[C]// Thirty-Second AAAI Conference on Artificial Intelligence. [S.l.]: [s.n.], 2018.

[18] CHEN H, DAI X, CAI H, et al. Large-scale Interactive Recommendation with Tree-structured Policy Gradient[J]. ArXiv preprint arXiv:1811.05869, 2018.

[19] LALMAS M, O'BRIEN H, YOM-TOV E. Measuring User Engagement (Synthesis Lectures on Information Concepts, Retrieval, and Services)[J]. San Rafael: Morgan & Claypool Publishers, 2014.

[20] SCHOPFER S, KELLER T. Long Term Recommender Benchmarking for Mobile Shopping List Applications using Markov Chains[J]. 2014.

[21] WU Q, WANG H, HONG L, et al. Returning is believing: Optimizing long-term user engagement in recommender systems[C]// Proceedings of the 2017 ACM on Conference on Information and Knowledge Management. ACM. [S.l.]: [s.n.], 2017: 1927–1936.

[22] SUTTON R S. Generalization in reinforcement learning: Successful examples using sparse coarse coding[C]// Advances in neural information processing systems. [S.l.]: [s.n.], 1996: 1038–1044.

[23] CAI H, REN K, ZHANG W, et al. Real-time bidding by reinforcement learning in display advertising[C]// Proceedings of the Tenth ACM International Conference on Web Search and Data Mining. ACM. [S.l.]: [s.n.], 2017: 661–670.

[24] LIPTON Z C, BERKOWITZ J, ELKAN C. A critical review of recurrent neural networks for sequence learning[J]. ArXiv preprint arXiv:1506.00019, 2015.

[25] HOCHREITER S, SCHMIDHUBER J. Long short-term memory[J]. Neural computation, 1997, 9(8): 1735–1780.

[26] CHO K, VAN MERRIËNBOER B, BAHDANAU D, et al. On the properties of neural machine translation: Encoder-decoder approaches[J]. ArXiv preprint arXiv:1409.1259, 2014.

[27] TESAURO G. TD-Gammon, a self-teaching backgammon program, achieves master-level play[J]. Neural computation, 1994, 6(2): 215–219.

[28] SUTTON R S, MCALLESTER D A, SINGH S P, et al. Policy gradient methods for reinforcement learning with function approximation[C]// Advances in neural information processing systems. [S.l.]: [s.n.], 2000: 1057–1063.

[29] DEISENROTH M P, NEUMANN G, PETERS J, et al. A survey on policy search for robotics[J]. Foundations and Trendső in Robotics, 2013, 2(1–2): 1–142.

[30] WEAVER L, TAO N. The optimal reward baseline for gradient-based reinforcement learning[C] / / Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence. Morgan Kaufmann Publishers Inc. [S.l.]: [s.n.], 2001: 538–545.

[31] WILLIAMS R J. Simple statistical gradient-following algorithms for connectionist reinforcement learning[J]. Machine learning, 1992, 8(3-4): 229–256.

[32] SUTTON R S, SZEPESVÁRI C, GERAMIFARD A, et al. Dyna-style planning with linear function approximation and prioritized sweeping[J]. ArXiv preprint arXiv:1206.3285, 2012.

[33] LI L. Offline evaluation and optimization for interactive systems[J]. 2015.

[34] THOMAS P. Bias in natural actor-critic algorithms[C] / / International Conference on Machine Learning. [S.l.]: [s.n.], 2014: 441–448.

[35] ÅSTRÖM K J. Optimal control of Markov processes with incomplete state information[J]. Journal of Mathematical Analysis and Applications, 1965, 10(1): 174–205.

[36] KAELBLING L P, LITTMAN M L, CASSANDRA A R. Planning and acting in partially observable stochastic domains[J]. Artificial intelligence, 1998, 101(1-2): 99–134.

Acknowledgements

I'd like to thank my advisor Profeesor Weinan Zhang for letting me work with him on this topic. Advice given by him has been a great help in this thesis project;
I am particularly grateful for the assistance from my thesis mentor Xinyi Dai in the Apex lab. Her explanations, suggestions and guidance on this thesis project has been very much appreciated;
I wish to acknowledge the help from Haokun Chen in the Apex lab for answering so many of my project-related questions. His paper in RL for recommendation system has been a great source of assistance for this project;
My special thanks are extended to Han Wang and Yangchen Pan from the RLAI lab in the University of Alberta for their useful suggestions on the model-based RL part of this thesis;