# Learning to Recommend using a Deep Reinforcement Agent

*Hernan Santiago Gonzalez Toral*

*Department of Computer Science*
*University College London*

**Supervisor**
*Dr. Jun Wang*
*jun.wang.l@cs.ucl.ac.uk*

This report is submitted as part requirement for the
**MSc in Web Science & Big Data Analytics**
at
**University College London**.
It is substantially the result of my own work except
where explicitly indicated in the text.

Department of Computer Science
University College London

August 30, 2016

**This page is purposely left blank.**

# Abstract

My research is about stuff.

It begins with a study of some stuff, and then some other stuff and things.

There is a 300-word limit on your abstract.

**This side is purposely left blank.**

# Acknowledgements

Acknowledge all the things!

**This side is purposely left blank.**

# Contents

# List of Figures

# List of Tables

## Chapter 1

# Introduction

## 1.1 Problem Overview

The volume of information available on the internet have been increasing rapidly with the explosive growth of the World Wide Web, e-Commerce and other online services, making difficult for users to find relevant products in a short period of time. To avoid this problem, many websites use recommendation systems to help customers to find items that satisfies their needs[4]. Suggestions for books on Amazon, or movies on Netflix, are real-world examples of Recommender Systems (RS) that demonstrate that the evolution of RS and the web should go hand-in-hand. As a result, the final goal of a RS is to efficiently manage an ever growing set of items and generate meaningful recommendations to a collection of users with common interests.

The design of such recommendation engines depends on the domain and the particular characteristics of the data available. While *Collaborative Filtering* systems analyze historical interactions alone, *Content-based* RS systems are based on user profile attributes; and finally, *hybrid* techniques combine both of these designs. Latest studies in CF[1] showed that combining conceptual and usage information can improve the quality of web recommendation.

Current RS typically act in a *greedy* manner by recommending items with the highest user ratings. However, greedy recommendations are suboptimal over the long term. This approach does not actively gather information on user preferences and fails to recommend novel songs that are potentially interesting. A successful recommender system must balance the need to explore user preferences and to exploit this information for recommendation.

Therefore, the architecture of recommender systems and their evaluation on real-world problems is an active area of research. In fact, RS have found an active application area for diverse Information Retrieval, Web Mining and Machine Learning techniques[1]. Reinforcement learning[2] for example, has been suited to solve the recommendation problem. In general, once a user makes her choice based on a list of recommendations given by a RS, a new list of recommended items is then presented. Thus, the recommendation process can be thought as a sequential process, where users choices are sequential by nature (e.g. a user buy a book by the author of a recent book we liked).

Indeed, reinforcement learning has demonstrated to be a potentially effective approach in the domain and particularly good to solve the cold-start problem and in some extent, the exploration/exploitation trade-off[5]. Nevertheless, it has received relatively little attention and found only limited application.

On the other hand, most recent research[6] have focused on making a generalization of recent advances in Deep Learning[7] from Identically and Independently Distributed (i.i.d.) input to the non-i.i.d. (CF-based) input, and propose hierarchical Bayesian models that jointly performs deep representation learning for the content information and CF for estimating the ratings matrix. Experiments carried out shown that Deep Learning approaches can significantly outperform the state of the art. As a result, current research has been getting involved on using deep representational approaches that can lead to better recommendations.

Furthermore, a recent work presented by Dulac-Arnold et al. on deep reinforcement learning in large discrete action spaces[8] presented a new policy architecture that not only allows reinforcement learning methods to be applied to large-scale learning problems, and to operate efficiently with a large number of actions, but also can generalize over the action set in logarithmic time. This algorithm showed to a promising convergence of the model among different continuous environments, where a simulated recommender system was also considered.

The main objective of this project is to show that a deep reinforcement learning model is able to perform well under a recommendation system environment and generate good recommendations to users over a large set of items. After defining a simulated recommendation system environment under the Open AI gym framework, the performance of the reinforcement agents based on the Deep Deterministic Policy Gradient algorithm with k-nearest neighbors (kNN) and Factorization Machine variants are compared. [***](Missing explanation of experiments and conclusions)

## 1.2 Dissertation Objectives and Structure

The remaining chapters of this report are organized as follows. Chapter 2 defines the recommender problem and introduces Reinforcement Learning. Following, it reviews some related work that has been made for modeling recommender systems using reinforcement learning approaches and deep learning. The proposed model and its implementation details are presented in Chapter 3, while the experimental evaluation and results are analyzed in Chapter 4. Finally, we present a summary of lessons learned and conclusions, and propose some ideas for future work.

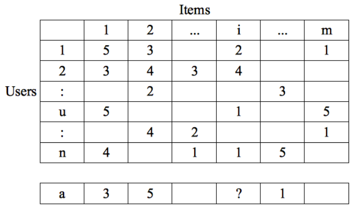
## Chapter 2

# Background

## 2.1 Recommender Systems

In recent years, there has been growing focus on the study of Recommender Systems (RSs), where different techniques have been published in order to to address the problem of information overload on the Internet. Consequently, the evolution of RS and the web usually go hand-in hand. Resnick and Varian[4] defined RS as systems that help users limit their search by supplying a list of items that might interest them. Such systems has found an active application area for diverse Information Retrieval, Web Mining and Machine Learning techniques that help to solve typical recommendation problems. Additionally, different RS mechanisms have been categorized depending on the way they analyze the data sources to find affinities between users and items.

The most general setting in which recommender systems are studied is presented in Figure 2.1. Having a rating matrix of n users and m items representing the current user preferences, the task of a RS is to predict all missing ratings $r_{a,i}$ for the active user a, and then recommend the item(s) with the highest rate[9]. However the user ratings matrix is typically sparse, as most users do not rate most items. Different approaches to solve this task can be categorized into the following types.



Items

| | | 1 | 2 | ... | i | ... | m |
|---|---|---|---|---|---|---|---|
| | 1 | 5 | 3 | | 2 | | 1 |
| | 2 | 3 | 4 | 3 | 4 | | |
| Users | : | | 2 | | | 3 | |
| | u | 5 | | | 1 | | 5 |
| | : | | 4 | 2 | | | 1 |
| | n | 4 | | 1 | 1 | 5 | |
| | a | 3 | 5 | | ? | 1 | |

**Figure 2.1:** Typical user ratings matrix, where each cell $r_{u,i}$ corresponds to the rating of user u for item i.

### 2.1.1 Collaborative Filtering

In Collaborative Filtering (CF) systems, items are recommended by exploiting the similarities amongst several users based on the feedback of previously consumed items. Usually, databases that fit into a CF approach, store the past users interactions in the form of explicit ratings (e.g., 1 to 5 range), or implicit ratings (e.g. a song played by a user, or an item bought by her). In general, CF methods are subdivided into: memory-based and model-based approaches.

- **Memory-based CF:** items are recommended using a weighted combination of ratings in a subset of users that are chosen based on their rating similarity to the target user. The most commonly used measure is the Pearson correlation coefficient[10]. Alternatively, the similarity between the ratings of two users (represented as a vector in an m-dimensional space) can be and computed using the Cosine similarity or the adjusted Cosine similarity measure which overcomes the issue of users with different rating behavior schemes[11]. Previous studies found that correlation[12] and adjusted cosine similarity[11] performs slightly better.

  Memory-based CF methods have been extended and improved over the years. Linden, Smith, and York[13] proposed an *Item-based* (or item-item) CF method to overcome the problem of dimensionality found when conventional memory-based algorithms cannot scale well when finding similarities between millions of users and items. This approach, which matches a user's rated items using Pearson correlation, lead to faster online systems and improved recommendations.

- **Model-based CF:** provides recommendations by using statistical models for predicting user ratings. The most widely used models are Bayesian classifiers, neural networks, fuzzy systems, genetic algorithms, latent features and matrix factorization[1]. For instance, CF methods can be turned into a classification problem, where a classifier is built for each active user representing items as features over users and available ratings as labels. However, latent factor and matrix factorization models have emerged as a state-of-the-art methodology in this class of techniques[14].

### 2.1.2 Content-based RS

Content-based (CB) RS (compared to pure CF that only utilizes the user rating matrix) tries to make a better personalized recommendation by exploiting the knowledge about a user (e.g. demographic information), or the properties of items (e.g. genre of a movie). Several approaches have treated this problem as an information retrieval (IR) task, where the content associated with the user's preferences is treated as a query, and the unrated items are scored with relevance/similarity to this query[15]. Alternatively, CB-RS has also been treated as a classification task using algorithms such as k-Nearest Neighbors (k-NN), decision trees, and neural networks[16]

### 2.1.3 Hybrid approaches

In order to leverage the strengths of both CB-RS and CF methods, several hybrid approaches have been proposed. For instance, Melville et al. presented in [17] a general framework for content-boosted collaborative filtering, where content-based predictions are applied to convert a sparse user ratings matrix into a full ratings matrix, and then a CF method is used to provide recommendations. In essence, this approach has been shown to perform better than pure CF, pure CB-RS, and a linear combination of the two.

### 2.1.4 The cold-start problem

The cold-start problem is a common issue that happens in recommender systems when it is not possible to make reliable recommendations due to an initial lack of ratings. There are three kinds of known cold-start problems: new community, new item and new user[1]. However, the latter represents one of the greatest difficulties faced by an RS in operation. Since new users have not yet provided any rating in the RS, they cannot receive any personalized recommendations when using memory-based CF. Usually, when the users enter their firsts ratings they expect the RS to offer them personalized recommendations, but there are not enough ratings yet to be able to make reliable predictions. As a consequence, new users feel that the RS does not offer the service they expected.

This problem is often faced using hybrid approaches such as CF-CB RS, CF-demographic based RS, or CF-social based RS[1]. However, these methods can be combined with clustering techniques over items in order to improve the prediction quality.

### 2.1.5 Trends in Recommendation Systems

Latest studies in CF showed that combining conceptual (explicit) and usage (implicit) information can improve the quality of web recommendations. Bobadilla et al.[1] presented a taxonomy for RS which unifies the current recommender methods and algorithms that can be applied to incorporate memory-based, social and content-based information into their CF system depending on the type of information available. The taxonomy depicted in Figure 2.2, also detail at its higher levels the current evaluation methods for RS in terms of quality measures, diversity and novelty.

Moreover, Bobadilla et al. argue that one the most widely used algo-

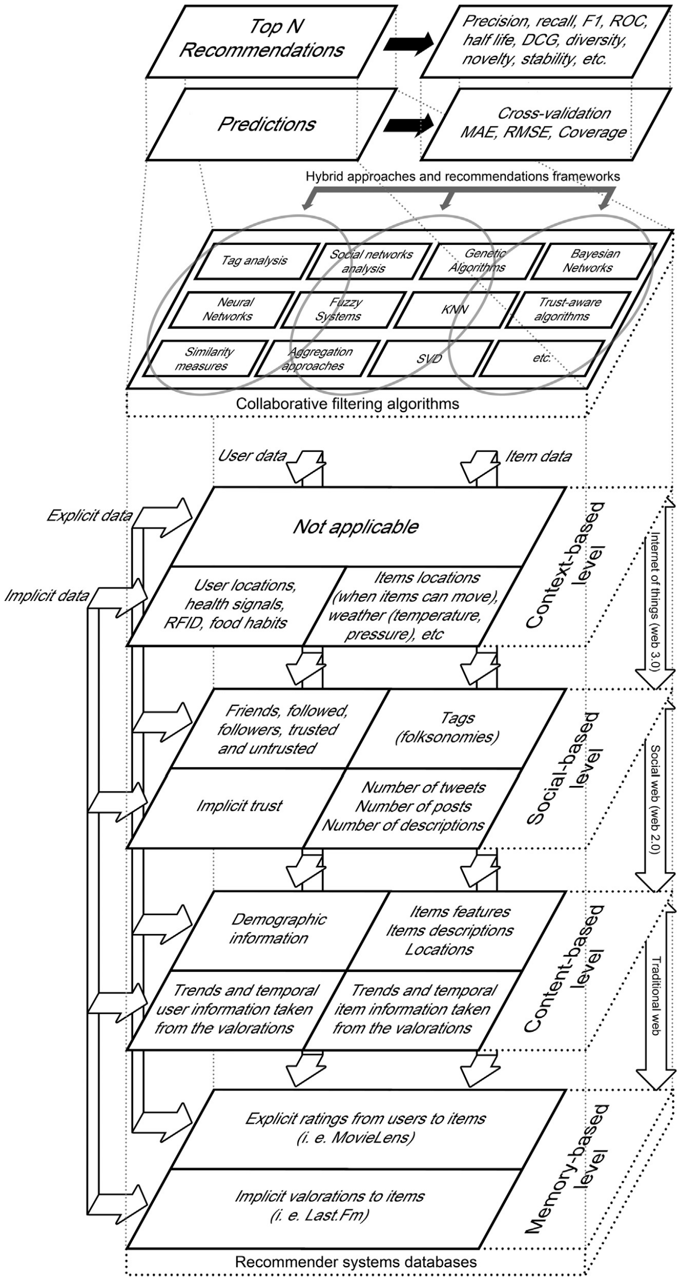

Top N Recommendations
Precision, recall, F1, ROC, half life, DCG, diversity, novelty, stability, etc.
Predictions
Cross-validation MAE, RMSE, Coverage
Hybrid approaches and recommendations frameworks
Tag analysis
Social networks analysis
Genetic Algorithms
Bayesian Networks
Neural Networks
Fuzzy Systems
KNN
Trust-aware algorithms
Similarity measures
Aggregation approaches
SVD
etc.
Collaborative filtering algorithms
User data
Item data
Explicit data
Implicit data
Not applicable
User locations, health signals, RFID, food habits
Items locations (when items can move), weather (temperature, pressure), etc
Context-based level
Internet of things (web 3.0)
Friends, followed, followers, trusted and untrusted
Tags (folksonomies)
Implicit trust
Number of tweets Number of posts Number of descriptions
Social-based level
Social web (web 2.0)
Demographic information
Items features Items descriptions Locations
Trends and temporal user information taken from the valorations
Trends and temporal item information taken from the valorations
Content-based level
Traditional web
Explicit ratings from users to items (i. e. MovieLens)
Implicit valorations to items (i. e. Last.Fm)
Memory-based level
Recommender systems databases

**Figure 2.2:** Recommender System taxonomy. Source: Recommender systems survey[1]

rithm for CF is the k-nearest neighbors (kNN) model. In general, kNN generates recommendations by executing the following tasks: (1) determine k users neighbors; (2) implement an aggregation approach with the ratings for the neighborhood in items not rated by a; and (3) select the top N recommendations from predictions obtained in step 2.

On the other hand, a more robust model that combines the advantages of Support Vector Machines with factorization models was introduced by Rendle S. in [3]. The Factorization Machine (FM) model is able to compute all interactions between variables using factorized parameters even in scenarios with huge sparsity like recommender systems. Later, Rendel S. presented in [18] the *LibFM* software package that implements three learning methods have been proposed for FMs: Stochastic Gradient Descent (SGD)[3], Alternating Least-Squares (ALS)[19] and the Markov Chain Monte Carlo (MCMC)[20].

Most recent research like [6], made a generalization of recent advances in Deep Learning [7] of the Identically and Independently Distributed (i.i.d.) input and applied it to the non-i.i.d. (e.g. CF-based) input by presenting hierarchical Bayesian models that jointly performs deep representation learning for the content information and CF for the ratings matrix. Additionally, Wu Y. et al. in [21] proposed a Collaborative Deep Auto Encoders (CDAE) model which formulates the top-N recommendation problem using Denoising Auto-Encoders to learn the latent factors from corrupted inputs. Experiments carried out over different domains have shown that the two deep learning approaches can significantly outperform the state of the art. Therefore, recent research has started to focused on defining deep representational approaches that can lead to better recommendations.

## 2.2 Reinforcement Learning

Reinforcement learning (RL) [22] is a family of machine learning algorithms that optimize sequential decision making processes based on scalar evaluations or rewards. Similarly to an n-armed bandit model[23], RL considers the problem as a goal-directed agent interacting with an uncertain environment, where the main objective is to perform actions that maximize the expected sum of future reward for each state in the long term. The most important feature distinguishing RL from other types of learning is that it uses training information that evaluates the actions taken rather than instructs by giving correct actions (like supervised learning algorithms). In other words, an agent must be able to learn from its own experience, like a trial-and-error search.

RL uses a formal framework which defines the continuous interaction in terms of states, actions, and rewards, between the *agent* and an *environment*[2]. At each time step $t$, the agent receives some state representation of the environment $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of possible states. Then it selects an action $a_t \in \mathcal{A}(s_t)$ , where $\mathcal{A}(s_t)$ is the set of actions available in the observed state. One time step later, the agent receives a numerical reward $r_{t+1} \in \mathcal{R}$, and the environment turns into a new state $s_{t+1}$. Figure **??** shows the whole interaction in an agent-environment interface.

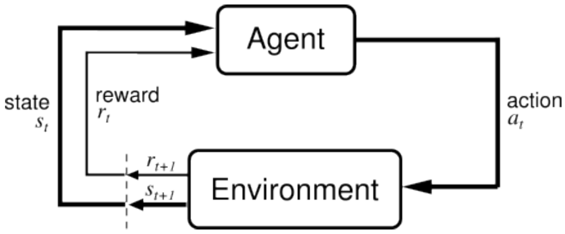


**Figure 2.3:** The Agent-Environment interface in a Reinforcement Learning problem. Source: Reinforcement Learning: a Survey[2]

This framework is intended to be a simple way of representing four essential features of the artificial intelligence problem: (1) a *policy* (commonly stochastic) defines a mapping from the perceived states of the environment to actions to be taken when in those states; (2) a *reward* function maps each state-action pair to a single number that represents how good or bad the new state is for the environment; (3) a *value* function specifies the total amount of reward an agent can expect to accumulate over the future and starting from a given state. This function is considered the most important as it is used by the agent during decision-making and planning; and optionally (4) a model that mimics the behavior of the environment (transitions and rewards) and provides a way of deciding which action to perform considering possible future situations before they are actually experienced.

In general, agents are categorized depending on how the reinforcement learning problem needs to be modeled. *Value-based* and *Policy-based* agents are solely based on the value and policy functions respectively. *Actor-critic* agents use both policy and value functions; *Model-free* agents use policy and/or value function but no model; and *Model-based* agents have all properties mentioned above.

On the other hand, The *return* $R_t$ is defined as the sum of the discounted future rewards over an episode of $T$ time steps that an agent actually seeks to maximize:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.1}$$

where $\gamma, 0 \leq \gamma \leq 1$ is a discount factor that determines the present value of future rewards

### 2.2.1 Markov Decision Processes

A Markov Decision Process (MDP) is a model for sequential stochastic decision problems. As such, it is widely used in applications where an autonomous agent is influencing its surrounding environment through actions[5]. It is defined by a tuple $\langle S, A, R, Pr\rangle$ representing a Markov Chain with values and decisions that follows the Markov property: *a state S is Markov if and only if* $\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, ...S_t]$*. In other words, in a MDP the future is independent of the past given the present.* Therefore, if a reinforcement learning task satisfies the Markov property and the state and action spaces are finite, then it can be modeled as a finite Markov Decision Process.

A particular finite MDP is defined by its state and action sets, a reward function $R$ (equation 2.2) that assigns a real value to each state/action pair, and a state-transition function $Pr$ (equation 2.3) which provides the probability of transitioning between every pair of states given each action. Altogether, these quantities completely specify the most important aspects of the dynamics of a finite MDP.

$$\mathcal{R}^a_{ss'} = \mathbb{E}r_{t+1}|s_t = s, a_t = a, s_{t+1} = s' \tag{2.2}$$

$$\mathcal{P}^a_{ss'} = Prs_{t+1} = s'|s_t = s, a_t = a \tag{2.3}$$

Therefore, the decision-maker's goal is to find an optimal policy $\pi$, such that at each stage of the decision process, the agent needs only to obtain the current state $s$ and execute the action $a = \pi(s)$. Various exact and approximate algorithms have been proposed for estimating the optimal policy $\pi*$, such as policy iteration, value iteration, Monte-Carlo learning, temporal difference, Q-learning, SARSA, etc[22][2].

### 2.2.2 A model-free Reinforcement Learning Setup

A reinforcement learning setup consists of an agent interacting with an environment $E$ at discrete time steps. At each time step $t$, the agent receives an observation $s_t$, takes an action $a_t$ and receives a reward $r_t$. Additionally, the environment $E$ may be stochastic so it can be modeled as an MDP with a state space $\mathcal{S}$, action space $\mathcal{A}$, an initial state distribution $\rho(s_1)$, transition dynamics $\rho(s_{t+1}|s_t, a_t)$, and reward function $r(s_t, a_t)$. On the other hand, the agent's behavior is defined by a policy $\pi$, which maps states to a probability distribution over the actions $\pi : \mathcal{S} \rightarrow P(\mathcal{A})$. Finally, the return from a state is defined as the sum of the discounted future reward $R_t = \sum_{i=t}^{T} \gamma^{(i?t)} r(s_i, a_i)$. As the return depends on the actions chosen and therefore on $\pi$, it may be stochastic.

The goal in reinforcement learning is to learn a policy which maximizes the expected return from a start distribution $J = \mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi}[R_1]$. The action-value function is used in many RL algorithms and describes the expected return after taking an action $a_t$ in state $s_t$ and thereafter following policy $\pi$:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_{i \geq t}, s_{i>t} \sim E, a_{i>t} \sim \pi}[R_t | s_t, a_t] \tag{2.4}$$

Many approaches in RL opt to represent the action-value function as a recursive relationship using the Bellman equation (eq. 2.5). Nevertheless, If the target policy is deterministic we can describe it as a function $\mu : \mathcal{S} \leftarrow \mathcal{A}$ and avoid the inner expectation as shown in equation 2.6. As the expectation depends only on the environment, it is possible to learn $Q^{\mu}$ off-policy, using transitions which are generated from a different stochastic behavior policy $\mu$.

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim \pi}[Q^{\pi}(s_{t+1}, a_{t+1})]] \tag{2.5}$$

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu_{t+1})] \tag{2.6}$$

Q-learning[24], is a commonly used off-policy algorithm which uses a *greedy* policy $\mu(s) = argmax_a Q(s, a)$ in order to estimate the action that gives the maximum reward. As the algorithm considers function approximators parameterized by $\theta^Q$, it can be used as an optimizer that minimize the loss:

$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^{\beta}, a_t \sim \beta, r_t \sim E}[(Q(s_t, a_t | \theta_Q) - y_t)^2] \tag{2.7}$$

where $y_t = \mathbb{E}_{s' \sim \mathcal{E}}[r(s_t, a_t) + \gamma max_{a'} Q(s_{t+1}, \mu(s_{t+1}) | \theta^Q)]$.

### 2.2.3 Exploration/Exploitation Dilemma

One of the challenges that arise in reinforcement learning is the trade-off between exploration and exploitation[22][2]. To obtain a lot of reward, a reinforcement learning agent must prefer actions that has already tried in the past and found to be effective in terms of reward. In order to discover such patterns, it needs to try actions that it has not been selected before. so the agent has to be able to exploit actions that are already known as effective, but it also needs to explore in order to make better action selections in the future.

Naïve approaches use a greedy policy $\pi(s) = argmax_a Q(s, a)$ to get the action with maximum reward as a pure exploitative model. However, an $\epsilon$-greedy policy can be considered instead to allow exploration in the environment and improve the estimation of the non-greedy action values. Therefore, the agent will select a random action with a probability $\epsilon$, and will act greedily with probability $1 - \epsilon$ . The advantage of $\epsilon$-greedy policy over greedy policy is that the former continuously explore and improve the chances of recognizing possible actions that can lead to better rewards.

Nevertheless, this has still been an issue to solve in some reinforcement learning tasks, so research has continuously focused on trying different techniques to solve the exploration/exploitation trade-off that can lead to the discovery of better policies[22].

## 2.3 Recommender Systems and Reinforcement Learning

Although reinforcement learning seems to have great potential for improving the recommendation problem, it has received relatively little attention and found only limited application. The first experiments on using reinforcement learning techniques were focused on developing recommendation systems for web content using implicit data from server files which stored the navigational logs of the users throughout their contents.

Ten Hagen et al. [25] considered RL approach with an exploration/exploitation trade-off for discovering unknown areas and automatically improve their recommendation policy to helping users navigate through an adaptive web site. They stated that a model-free algorithm like Q-Learning can become infeasible if the number of actions are relatively high and showed that a greedy policy can indeed lead to a suboptimal recommendation model as the use of such kind of policy function, where the b̈estäctions (e.g with higher probability) are taken, does not always improve the policy. They finally demonstrated that this problem is solved by starting the algorithm with a lot of exploration and gradually reduce the parameter $\epsilon$ as the policy is getting closer to an optimum. Results concluded that a recommender system without exploration potentially can get trapped into a local maxima.

Rojanavasu et al. in [26] presented a general framework for web recommendation that learns directly from customer's past behavior by applying an RL process based on the SARSA method and a $\epsilon$-greedy policy. The system was composed of two models: a global model to keep track of customers trends as a whole, and a local model to record the user's individual browsing history. In general, the model treated pages as states of the system and links within a page as actions. To predict the next state, it used a ranking system that is also separated into 2 parts. i) a global ranking system using the data from a $Q_{global}$-matrix obtained from the action-value function and an $\epsilon$-greedy policy that gave the chance for rank new items that have few clicks but may match a user's interest; and ii) a local ranking $Q_{local}$ using an inverse $\epsilon$-greedy policy. The system then finds the final total reward using $Q_{total} = Q_{local} + wQ_{global}$ where $w \in (0-1]$ is a weight hyper parameter of the model.

Overall, the system provided customers the chance to explore other products than the ones they already visited. Experimental results showed that a value of $\epsilon = 0.2$ preserves the balance between exploration and exploitation, meanwhile If $\epsilon < 0.2$ the system will gave small chances to users to explore new items, or otherwise may include products that do not match to the customer's interest ($\epsilon > 0.2$). On the other hand, results also prove that even if the purpose of $Q_{local}$ was to discover new products, it appeared to be less effective than $Q_{global}$.

Shani et al. in [5] argued that it is more appropriate to formulate the problem of generating recommendations as a sequential optimization problem so they described a novel approach for a commercial web site based on MDPs together with a predictive model. The MDP-based recommender system took into account the expected utility and the long-time effect of a particular recommendation. Then it suggested items whose immediate reward is lower, but leads to more *profitable* rewards in the future. However, the benefits of this model are offset by the fact that the model parameters are unknown, randomly initialized and that they take considerable time to converge. So, they defined a strong initial model-based for collaborative filtering, which solves quickly, and that does not consume too much memory.

Before implementing the recommender, they initialize a predictive model of user behavior using data extracted from the web site, and then used it to provide the initial parameters for the MDP. They proposed to use maximum-likelihood estimation with three major enhancements: skipping, clustering, mixture modeling, so the predictive model can be thought of as a first-order Markov chain (MC) of user dynamics in which states correspond to sequences of $k$ events (in this case: previous selections) representing relevant information about the users interaction. Then, the transition function described the probability that a user ,whose $k$ recent selections were $x_1, ..., x_k$ will select the item $x'$ next.

Authors experimented with small values of $k$ ($k \in [1-5]$) in order to overcome data sparsity and MDP solution complexity issues. Moreover, enhancements on the maximum-likelihood n-gram formulation showed to be useful to find a good estimate of the transition function for the initial predictive model without suffering the problem of data sparsity and bad performance presented on other model variations.

On the other hand, the components of the reinforcement learning model were defined as follows: a) the states are the $k$-tuples of items purchased; b) actions correspond to a recommendation of an item; c) rewards depends only on the last item defining the current state, where its net profit was used; and d) the transition function as the stochastic element of the model that estimates the user's actual choice.

In order to solve the MDP problem, a policy iteration algorithm was used as the defined state space presented certain characteristics that lead to fast convergence. The transition function was carefully initialized in order to be fairly accurate when the system was first deployed to avoid the cold-start problem,causing states that are infrequently observed to be updated faster than already observed states. Moreover, when the first transition to a state is observed, its probability is initialized to 0.9. In this way, the system balances the need to explore unobserved items in order to improve its model by recommending non-optimal items occasionally until getting their actual counts. Finally, to update the model, the system used an off-line approach which keeps track of the recommendations and the user selections and build a new model at fixed time intervals (e.g. once a week).

Experiments demonstrated that a deployed system using three mixture components (combined using an equal weight), with history length $k \in [1, 3]$ was able to generate 28% higher average user reward compared to pure MC model, while in terms of computational costs, the MDP-based model was built quickly and provided fastest recommendations at the price of more memory use. All in all, the off-line predictive model approach provided an adequate initial performance that overcomes the cold-start problem, however, authors avoid using some form of reinforcement learning technique as they argued that at that time, its implementation requires many calls and computations by a recommender system online, which lead to slower responses and undesirable results for the web site owner.

A machine learning perspective, introduced by Taghipour et al. in

[27], used an off-line reinforcement learning model to solve the recommendation problem using the Q-Learning algorithm while employing concepts and techniques commonly applied in the web usage mining domain. They argued that this method is appropriate for the nature of web page recommendation problem as it provides a mechanism which is constantly learning, does not need periodic updates, can be easily adapted to changes in the website structure and new trends in users behavior.

The RL problem formulation was considered as as a competition between different recommender systems to gather more points, together with a stochastic model with the following properties:

- *states:* showing the history of pages visited by the user so far. For this case, a notion of N-Grams was adopted and a sliding window of size $w$ was set to limit the page visit sequences to a constant number and avoid large state spaces.

- *actions:* consisting of a single page recommendation at each state.

- *policy:* rewards actions positively if it recommends a page that will be visited in one of the consequent states

- *reward* function: defined as $r(s, a) + = reward(Dist(Rs', p), Time(p^v_w))$, where $Dist(Rs', p)$ is the distance of page p from the end of the recommended pages list to state $s'$, and $Time(p^v_w)$ indicates the time user has spent on the last page of the state. Finally, the $reward(Dist, Time)$ function was defined as a linear combination of both values: $reward(Dist, Time) = \alpha \times dist + \beta \times Time$ with $\alpha + \beta = 1$.

The proposed off-line RL method used a simplified formulation of the recommendation problem but it obtained much better results compared with two baselines methods: association rules and item-based collaborative filtering (with probabilistic similarity measure). As the coverage increase, naturally accuracy decrease in all systems, but the RL approach outperformed the other two systems displaying a lower rate in which its accuracy decreased. Authors concluded that the algorithm is a good candidate for solving the recommendation problem as it does not rely on any previous assumptions regarding the probability distribution of visiting a page after having visited a sequence of pages, and that the nature of the problem matches perfectly with the notion of delayed reward (known as temporal difference). Additionally, they suggested that in order to produce better results it could be possible to use a more complicated formulation of the reward function such as a neural networks, rather than a linear combination of factors.

Later in [28] they exploited the previous RL framework and present a hybrid web recommendation method enriched with semantic knowledge about the usage behavior and thus obtain a more generalized solution regarding to the usage data it has. The new system used the incremental Document Conceptual Clustering[29] algorithm to map pages to higher level concepts, and thus exploit the hierarchical and conceptual document clustering to provide a semantic relationship between them in combination with the usage data in a user session.

Therefore, new states consist of a sequence of concepts visited by the user, while actions are recommendation of pages that belong to a specific concept. This definition resulted in a much smaller state-action space as the size of the state space is now dependent on the number of distinct page clusters. Finally, the new reward function takes into account the content similarity of the recommended and visited pages along with the usage-based reward defined in the previous approach.

The modified algorithm do not make predictions based on weak usage patterns as the new states represented a generalized view of many single visit sequences. Furthermore, evaluation results showed the flexibility of the new RL approach to incorporate different sources of information in order to improve the quality of recommendations.

A model-based RL agent for music playlist recommendation proposed by Liebman et al. [30] modeled the preferences of both songs and song transitions as MDPs, and demonstrated to be potentially effective in the domain and particularly good to solve the cold-start problem as it is able to generate personalized song sequences within a single listening session and with no prior knowledge of the new user's preferences.

The adaptive playlist generation problem was defined as an episodic MDP $\langle S, A, P, R, T \rangle$ with the following components:

- a state space S composed by the ordered sequence of songs played, $S = \{(a_1, a_2, ..., a_i)|1 \leq i \leq k; \forall j \leq i, a_j \in \mathcal{M}\}$

- an actions space A formed by all possible candidates for being the next song to play, $a_k \in A$ which means $A = \mathcal{M}$

- a deterministic transition function $P(s, a) = s'$ which indicates the existing transitions from state s to s' by taking action a

- an utility function R(s, a) derived from hearing song a when in state s

- $T = \{(a_1, a_2, ..., m_k)\}$: the set of playlists of length k.

In order to find an optimal policy $\pi*$ (and thus obtain the most pleasing sequence of songs to the listener), a model-based approach was chosen arguing that even though model-free approaches learn the value of taking an action a from state s directly, it requires a lot of data to converge, and it is considerably scarce in the domain music recommendation (as well as in other kind of applications). On the other hand, model-based formulations often requires a lot of computation to find an approximate solution to the MDP, but the trade-off of computational expense for data efficiency makes the model-based approach a good option for this problem.

Since P is deterministic, only the listener's reward function R was required to be modeled. Therefore, the reward function defined as $R(s,a) = R_s(a) + R_t(s,a)$ represented the sum of two distinct components: (1) the listener's preference over songs, $R_s : A \rightarrow \mathcal{R}$, and (2) the preference over transitions from songs played to a new song, $R_t : S \times A \rightarrow \mathcal{R}$.

In essence, the model represents each song as a compacted vector of spectral auditory descriptors that capture meaningful differences in user's preferences, so the system is able to leverage knowledge using only a few transition examples to plan a future sequence of songs. On the other hand, the agent architecture was composed by a module for learning the listener parameters during initialization and learning on the fly processes; and an additional module for planning a sequence of songs in the playlist.

The initialization is divided in two parts: (1) initialization of song preferences polls the listener for her $k_s$ favorite songs in the database and updates the user's preference vector using the feature descriptors of each of the selected items; and 2) initialization of the transition preferences by presenting different possible transitions that encapsulate the variety in the dataset, and directly asking which of a possible set of options the listener would prefer. Updates are carried out in the same manner as the initialization of song preferences.

After initialization, the agent begins to play songs for the listener, waiting for her feedback (reward), and updating the user's preferences accordingly. This learning of the fly procedure is perceived as a temporal-difference update with an attenuating learning rate that balances the trust between the previous history of observations and the newly obtained signal. Then, a Monte Carlo Tree Search (MCTS) heuristic[31] is used for planning. The iterative process chooses a subset of 50% of the songs in the database with highest $R_s$ score, and then at each point, it simulates a trajectory of future songs selected at random and calculate the expected payoff of the song trajectory using $R_s$ and $R_t$. The process ends when it finds the trajectory which yields to the highest expected payoff, and finally, the first item in the trajectory is selected to be the next song to recommend.

Additionally, to mitigate the problem of high complexity during replanning under large song spaces, the agent used the canonical K-means algorithm for clustering songs according to song types and reduce the search complexity drastically.

An evaluation of the cumulative reward distribution of the proposed approach showed that overall, the DJ-MC agent outperforms two baselines models (a random agent and a greedy agent), while in terms of transition reward preferences, the new system presented a small but significant boost in performance compared to a model based only on reasoning about song preferences. All in all, this approach demonstrated to be a good improvement as a reinforcement learning model for music recommendation, as it enables learning from relatively few examples, and provides quality on recommendation sequences.

## 2.4 Deep Reinforcement Learning

So far, the RL agents presented above have shown to achieve some success under the recommendation domain, nevertheless, their performance has been conditioned to the quality of the hand-crafted feature engineering made, as well as the custom definition of value functions or policy representations. Moreover, to use RL successfully in situations approaching to real-world complexity, agents usually need to derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalise past experiences to new situations.

Recent advances in deep learning have made it possible to extract high-level features from raw sensory inputs[7]. Even if it seems that this approach could fit in a RL task, its applicability to the RL domain may present several challenges from a deep learning perspective, For instance, successful deep learning applications have required large amounts of hand-labelled training data to obtain good predictions results. Additionally, most deep learning algorithms assume the data samples to be i.i.d., while in RL an agent typically encounters with non-i.i.d. sequences of highly correlated states.

On the other hand, agents must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed, while the RL data distribution changes as the algorithm learns new behaviour. However, thanks to the increasing success of adapting different deep learning techniques to different domains, Deep Reinforcement Learning (DRL) models have started to be proposed, getting promising results.

Mnih et al. presented in [32] [33] the first deep learning model which uses an end-to-end reinforcement learning approach to learn control policies directly from a high-dimensional raw input space. The model, applied to a range of Atari 2600 games, consists of a convolutional neural network and an experience replay mechanism[34] to alleviate the problems of correlated data and non-stationary distributions in typical RL problems.

Moreover, the model's architecture uses only the state representation as input, and a separate output unit for each possible action (target network), corresponding to the predicted Q-values of each individual action that can be applied to the input state. This configuration allows the computation of all possible actions reward in a given state with only a single forward pass. Finally, the resulting Deep Q neural network (DQN) is trained with a variant of Q-learning which learns from raw pixels input and the underlying environment properties, and outputs a value function estimating the future rewards.

On the other hand, the RL formulation models the environment $\mathcal{E}$ as a MDP composed by a state space $\mathcal{S}$, a discrete action space $\mathcal{A} = \{1...K\}$, a scalar reward function $r(s_t, a_t)$ and a transition dynamics function $\rho(s_{t+1}|s_t, a_t)$. Then, given the agent's behaviour defined by a policy $\pi$, they estimate the action-value function by using an approximator function of the Bellman equation (presented in eq. 2.5) as an iterative update. A Q-network function approximator with weights $\theta$ is then trained by minimising the sequence of loss functions $L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}[(y_i - Q(s, a; \theta_i))^2]$ where $\rho(s, a)$ is the behaviour probability distribution over sequences of states s and actions a.

Finally, to compute the full expectations in the gradient of the loss function, they considered stochastic gradient mini-batch updates of the weight parameters, and a learning algorithm which is: a)*model-free*: replaces the expectations by directly using uniform samples from the behaviour distribution $\rho$, and b)*off-policy*: follows an $\epsilon$-greedy strategy of the behaviour distribution that ensures an adequate exploration of the state space.

Evaluation results showed that this method is able to learn how the value function evolves for a reasonably complex sequence of events. Furthermore, the proposed DRL algorithm demonstrated to perform better compared to the SARSA and Contingency[35] methods as they need to incorporate significant (handcrafted) prior knowledge about the visual problem to get considerable performance, in comparison to the raw inputs that DRL use. Also, DRL achieved better performance than an expert human player in 29 out of 49 games.

However, this model is only able to handle discrete low-dimensional action spaces and cannot be directly applied to tasks under the continuous domain as the iterative process of finding the action that maximises the action-value function becomes intractable. Additionally, a naïve discretization of the action space would no be a good solution as it may throw away information about the structure of the action domain.

Lillicrap et al. [36] adapted the ideas presented above to the continuous action space and defined an actor-critic, model-free approach, based on the Deterministic Policy Gradient[37] (DPG) algorithm, that is able to find policies end-to-end and whose performance is competitive compared to approaches based on a planning algorithm with access to the dynamics of the domain.

While the model's architecture uses similar features of DQN (network architecture, experience replay memory, and target outputs), along with batch normalization[38] to overcome the internal covariate shift problem in the network input layers during mini-batches, the Deep Deterministic Policy Gradient (DDPG) algorithm maintains two functions: (1)an actor function $\mu(s|\theta_\mu)$ that obtains the current policy by deterministically mapping states to specific actions; and 2)the critic function $Q(s,a)$ learned by applying the Bellman equation. During the learning process, DDPG updates the actor by applying the policy gradient to the expected return from a start distribution $J = \mathbb{E}_{r_i, si \sim E, a_i \sim \pi}[R_1]$ and with respect to the actor parameters. Even if convergence is no longer guaranteed due to the added non-linearity of the proposed approximation function, it produces a good generalization over large state spaces.

The DDPG algorithm also implements certain improvements to the RL task. First, it introduces stability in the learning process by applying soft target updated to the target networks instead of directly copying the weights. This is performed by exponentially moving their average values using $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ with $\tau \ll 1$. Second, batch normalization allows DDPG to be able to generalize across environments using the same hyper parameters for learning. Finally, the exploration-explotaition trade-off is managed independently from the learning algorithm by adding noise sampled from a noise process $\mathcal{N}$ to the actor policy $\mu$.

To evaluate the algorithm performance, authors used simulated physical environments with different levels of difficulty, and the Ornstein-Uhlenbeck[39] sampling process for adding exploration to the learning policy. Results demonstrated that even when harder tasks obtain poor Q estimates, DDPG is able to learn good policies across a variety of domains with continuous action spaces and using both low-dimensional feature vector and high-dimensional pixel inputs. Additionally, all the experiments were solved using fewer steps of experience than the DQN algorithm, showing that DDPG may be able to solve even more difficult problems.

However, the DDPG algorithm requires of a large number of training episodes to find solutions and only works under environments with a not so big number of actions. Therefore, a more robust model-free approach is needed in order to tackle these limitation.

Later, Dulac-Arnold et al. [8] presented a new policy architecture, under the same actor-critic framework used above, that not only allows reinforcement learning methods to be applied to large-scale learning problems, and to operate efficiently with a large number of actions, but also can generalize over the action set in logarithmic time. The full policy is then trained and optimized using DDPG. As a result, they obtained a more efficient algorithm that makes both learning and acting tractable in time and allows value-based policies to use the action features to reason about previously unseen actions.

The so-called *Wolpertinger* architecture leverages prior information about the actions and embed them in a continuous space upon which the actor can generalize using a smooth function approximator. After the policy produces a continuous action within this space, it uses an approximate nearest neighbour search to find the set of closest discrete actions in logarithmic time to finally select the action with the highest reward. Consequently, the algorithm avoids the heavy cost of evaluating all actions mainly by defining an efficient action-generating actor, and then using the critic to refine the actor's choices for the full policy.

Having the definition of the full policy as $\pi_{\theta^\pi} = g \circ f_{\theta^\pi}(s)$, where

$f_{\theta^\pi}(s) = \hat{\text{a}}$ is the proto-action generation function and $g$ the actual action function that return the set of k closest actions to â, the final goal of this approach is to perform policy iteration by alternatively performing policy evaluation on the current policy with Q-learning, and then improving upon the current policy by following the DDPG over $f_{\theta^\pi}$, considering that the effects of $g$ are a deterministic aspect of the environment.

The Wolpertinger agent, evaluated on three environment classes with large actions spaces (including a simulated recommender system), showed that it can scale to real-world MDPs with large number of actions as only a subset of the full set of actions was sufficient in many tasks to converge and provide significant speedups. Nevertheless, there is still an existing issue of exploration when the agent needs to learn from scratch.

## Chapter 3

# Models

This chapter will develop a reinforcement learning agent definition to solve the recommendation problem using deep learning techniques. After describing the main motivation behind this approach, details of the baseline model derived from previous work on the DDPG algorithm is presented, and finally, as the contribution to this MSc. project, a bayesian personalised ranking model based on factorization machines is introduced into the deep learning algorithm to try to enhance the policy architecture and therefore the system recommendations.

## 3.1 Motivation

Previously, reinforcement learning agents have received little attention to solve the recommendation tasks due to their action domain becomes intractable in current large-scale recommender systems. However, the great performance reached in several control tasks by end-to-end RL agents based on deep neural networks[33] have demonstrated the capacity to continuously adapt their behaviour without any human intervention and solve challenging human tasks. More interestingly, a single trained general-purpose deep reinforcement agent (using the same hyperparameters) was able to converge and perform well on several task environments with different action space definitions. Consequently, a new opened research area has been using recent breakthroughs in training deep neural networks to derive RL agents models that can adapt to other kind of existing tasks under both discrete and continuous domains and solve tasks that were previously considered intractable.

Another reason that motivates our approach is the sequential nature of the recommendation process. In general, recommender systems usually work in a sequential manner: once the RS suggest items to the user, she can accept one of the recommendations. At the next stage a new list of recommended items is calculated based on the user feedback and then presented again to the user. This sequential nature allows us to reformulate the recommendation process as a sequential optimization process where optimal recommendations can depend either on the previous items purchased, and in the order in which those items were purchased. For example, Zimdars et al. [40] suggested the use of a k-order Markov chain model (with k=3) to represent this behaviour by dividing a sequence of transactions $X_1, ..., X_T$ into cases $(X_{t-k}, ..., X_{t-1}, X_t)$ for $t = 1, ..., T$. and then, build a model to predict the item at the time step t given the other columns.

As now deep reinforcement learning approaches can converge to good estimations under tasks considered intractable before, it motivates us to evaluate and improve these models under recommendation tasks, which this thesis project is focused.

## 3.2 Baseline Agent Definition

Therefore, the recommender problem can be considered as a sequence of Markov Decision Processes $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma \rangle$, where the action space $\mathcal{A}$ is defined as the set of items to recommend, $\mathcal{S}$ represent the current state or item a user is currently consuming, thus both $\mathcal{S}$ and $\mathcal{A}$ represent items as feature vectors a and s $\in \mathbb{R}^n$ respectively. $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the transition probability distribution to move from state $s_t$ to $s_{t+1}$, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward value function $\mathcal{R}$ associated with an item accepted by the user, and $\gamma \in [0, 1]$ is the discount factor for future rewards.

The reinforcement learning agent is then defined as a model-free approach with the main objective to find the optimal policy $\pi$ that maximizes the expected total future reward $\mathbb{E}[R_1]$ over all episodes $R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i)$. The expectation can be obtained using the Bellman equation (presented in section 2.2.2):

$$Q^{\pi}(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \sum_{\mathbf{s}'} P(\mathbf{s}'|\mathbf{s}, \mathbf{a}) Q^{\pi}(\mathbf{s}', \pi(\mathbf{s}')) \tag{3.1}$$

In order to approximate $\pi$ and $Q$ under large-scale action spaces, standard actor-critic approaches need to define a policy by a parametrized actor function approximator $\pi_{\theta} = \arg\max_{\mathbf{a} \in \mathcal{A}} Q(\mathbf{s}, \mathbf{a}) : \mathcal{S} \to \mathcal{A}$ that would be able to generalize and that does not scales linearly in relation to the number of actions (turning the task into an intractable problem). Based on recent work presented by Lillicrap et al. **??** and Dulac-Arnold et al. [8], it is now possible to define generalizable reinforcement agents using a policy architecture with a sub-linear complexity relative to the action space that avoids the heavy cost of evaluating all actions. This actor-critic framework uses a multi-layer neural network as function approximators and is trained using Deep Deterministic Policy Gradient (DDPG).

The action generation function is based on the *Wolpertinger* policy defined in [8]. First, an initial function $f_{\theta^{\pi}}(s) = \hat{\mathbf{a}}$ provides a proto-action in $\mathbb{R}^n$ for a given state s. â is then mapped to the discrete action set $\mathcal{A}$ by applying the k-nearest-neighbor[1] function from equation 3.2, which returns the k closest actions in $\mathcal{A}$ by computing the $L_2$ distance between them.

$$g_k(\hat{\mathbf{a}}) = \arg\min_{a \in \mathcal{A}}^{k} |\mathbf{a} - \hat{\mathbf{a}}|_2 \tag{3.2}$$

Even if $g_k$ has the same complexity as the *argmax* in the Q action-

[1] The size of the generated action set k is task specific, and allows for an explicit trade-off between policy quality and speed

value function, each step of evaluation of the $L_2$ distance is faster than a full value-function evaluation as the lookup is performed in logarithmic time. Finally, the choice of the actual action to be applied to the environment is set according to the highest-scoring action obtained according $Q_{\theta^Q}$ defined in equation 3.3. As a result, the full Wolpertinger policy makes the algorithm significantly more robust to imperfections in the choice of action representation.

$$\pi_\theta(s) = \arg \max_{a \in g_k \circ f_{\theta^\pi}(s)} Q_{\theta^Q}(s, a) \tag{3.3}$$

On the other hand, the DDPG implementation maintains two functions: (1) the actor function $\mu(s|\theta_\mu)$ that obtains the current policy by deterministically mapping states to specific actions; and (2) the critic function $Q(s, a)$ learned by applying the Bellman equation. It then updates the actor by applying the chain rule to the expected return from a start distribution $J = \mathbb{E}_{r_i, si \sim E, a_i \sim \pi}[R_1]$ and with respect to the actor parameters:

$$\begin{aligned} \nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}] \\ &= \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_t}] \end{aligned} \tag{3.4}$$

In order to maintain stability in the learning algorithm, a separate target networks $\mu'(s|\theta_{\mu'})$ and $Q'(s, a|\theta_{Q'})$ for the actor and critic are used for calculating the target values, where their weight parameters are updated by using an smooth function update $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$ with $\tau \ll 1$ to main an exponential moving average of the trained parameters $\theta^\mu$ and $\theta^Q$.

Among the main advantages of using this deep learning approach, DDPG introduces an experience replay buffer of size N that helps to reduce the correlation among the sequence of samples used by the RL agent. The mechanism allows the behaviour distribution to be averaged over many of its previous states, smoothing out learning and avoiding instability or divergence in the parameters.

When executing the DDPG algorithm, the critic is trained from samples generated and stored in the replay buffer by the full policy, while the policy gradient $\nabla_a Q_{\theta^Q}(s, a)$ is taken at the true output â $= f_{\theta^\pi}(s)$. The target action in the Q-update is generated by the full policy.

## 3.3 FM Agent Model

Our implementation tries to replace the k-nearest neighbour computation in the Wolpertinger policy for a Bayesian Personalized Ranking (BPR) model using Factorization Machines to find the top-N items from a given state to then select the item that yields to the highest Q value in the long term. The deep reinforcement agent will finally learn using the same DDPG architecture as the baseline model.

The Bayesian formulation presented in Rendle et al. in [41] find the correct personalized ranking for all items $i \in I$ by maximizing the posterior probability $p(\theta| >_u) \approx p(>_u |\theta)p(\theta)$ where $\theta$ represents the parameter vector of an arbitrary model class (e.g. matrix factorization) and $>_u$ is the desired but latent preference structure for user $u$. In order to guarantee a personalized total order, totality, antisymmetry and transitivity properties are fulfilled by defining the individual probability that a user really prefers item $i$ to $j$ as $p(i >_u j|\theta) := \sigma(\hat{x}_{uij}(\theta))$ where $\sigma$ is the sigmoid function and $\hat{x}$ is an is an arbitrary real-valued function of the model parameter vector which captures the special relationship between user $u$, item $i$ and item $j$, such as Matrix Factorization of adaptive kNN.

Rendel S. showed in [18] that factorization models can be expressed using the Factorization Machines, including the BPR model for top-N recommendation, where vectors x are ordered by the score of $\hat{y}(x)$ and optimization is done over pairs of instance vectors $(x_i, x_j) \in \mathcal{D}$.

### 3.3.1 Proposed Approach

A new policy achitecture called FM-policy is proposed. It tries to improve how the agent learn to recommend under scenarios with high sparsity at the cost of linear complexity, compared to the sublinear complexity offered by the Wolpertinger policy. The FM-policy algorithm is described in Algorithm 1.

**Algorithm 1:** FM policy

```
State s previously received from environment.
$\hat{\mathbf{a}} = f_{\theta^\pi}(\mathbf{s})$ Receive proto-action from actor.
$\mathcal{A}_k = g_k(\mathbf{s})$ Retrieve Top-k recommendation given state s
$\mathbf{a}_{TOP} = \arg\max_{\mathbf{a}_j \in \mathcal{A}_k} Q_{\theta^Q}(\mathbf{s}, \mathbf{a}_j)$
if $\hat{\mathbf{a}} == \mathbf{a}_{TOP}$ then
    $updateBPRModel(\mathbf{s}, \mathbf{a})$
end
Apply $\mathbf{a}_{TOP}$ to environment; receive $r$, $\mathbf{s}'$
```

First, the actor function in the deep reinfocement learning architecture provides a proto-action in $\mathbb{R} \in A$ for a given state, while a pre-trained BPR-FM model obtains the the Top-k recommendations given the current state (or previous recommended item). Then, If the action predicted by the agent is the same as the item in list of k items that gives the highest Q value, the FM model is updated, and finally, the item that yields the highest reward is applied to the environment.

Similarly to the baseline model, when executing the DDPG algorithm, the critic is trained from samples generated and stored in the replay buffer by the FM policy, while the policy gradient is taken at the true output in the actor function $f_{\theta^\pi}(s)$. As a reference, the pseudocode of the algorithm can be found in Appendix 6.

## Chapter 4

# Experiments

In the previous chapter, deep reinforcement learning agents were described to solve the recommendation task. Now, in order to test and evaluate their performance, a simulated environment for Recommender Systems is going be defined. Then, after defining the baseline models and the experimental settings, two sets of experiments are carried out: (i) a performance evaluation of the proposed deep learning models and a pure Collaborative Filtering baseline model; and (ii) an analysis of the convergence and speed-up between the deep reinforcement learning agent based on previous work in the area, and an DRL agent with the new policy approach. Final conclusions on the experimental results obtained will be presented in the next chapter.

## 4.1 Reinforcement Learning Environment

For the purpose to demonstrate how the deep reinforcement learning agent perform on the recommendation task, a simulated environment is modelled under the OpenAI Gym[1] reinforcement learning toolkit. It that allow us to define Partially Observed MDP environments (POMDPs) under an episodic setting where the agent's experience is broken down into a series of episodes[42]. In general, OpenAI Gym manages two core concepts: (1) a free-to-code *agent* that maximizes the convenience for

[1] `https://gym.openai.com/`

users allowing them to implementation different styles of agent interface; and (2) a common *environment* and action/observation interfaces that ease the implementation and testing of different reinforcement learning problems under the same framework. The final goal in OpenAI Gym is to maximize the expectation of total reward per episode, and to achieve a high level of performance in as few episodes as possible.

During each episode, the agent's initial state is randomly sampled from a distribution, and the interaction proceeds until the environment reaches a terminal state. After performing an action step, the environment interface returns the observation of current state, the reward achieved by the action performed, a done flag indicating if an episode ends and an information object containing useful information for debugging and learning. Finally, the framework also allow us to measure the performance of a RL algorithm under an environment along two axes: *final performance* or average reward per episode, after learning is complete; and, *sample complexity* or the amount of time it takes to learn.

### 4.1.1 Recommender System Environment

The recommender system environment, characterized by a set of *n* items to recommend to *m* users, is defined (based on a previous definition in [8]) as a sequence of MDPs $\langle \mathcal{A}, \mathcal{S}, r, \mathcal{P} \rangle$ composed by an action set $\mathcal{A}$ that correspond to set of items to recommend, a state space $\mathcal{S}$ holding the item the user is currently consuming, a reward $r$ defined as the rating value given by a user to an item if she accepts it, and a transition probability matrix $\mathcal{W}$ which defines the probability of that a user will accept item *j* given that the last item she accepted was item *i*.

The transition probability matrix is generated using the ideas exposed by Yildirim et al. in [11] to build a random walk recommender system using a Markov Chain mode,l where the probability of being in a state only depends on the previous step $Pr(X_{u,k+1} = i|X_{u,k})$. The algorithm presented in Appendix A details the underlying process to set up the environment.

### 4.1.2 Simulation algorithm

The implemented algorithm is presented in Appendix B and describes the complete simulation process of the recommendation task. At each time-step, the agent presents an item $i$ to the user with action $\mathcal{A}_i$. The user then accepts the item according to the transition probability matrix $\mathcal{P}$ or selects a random item. To simulate the user patience in a session as in [8], an episode of learning ends with probability 0.1 if the user accepts the item, and with probability 0.2 if a random item is selected instead. Finally, after each episode the environment is reset by selecting a random item from a likely subset for an specific user provided by a single item-based collaborative filtering model based on the similarity between items.

## 4.2 Data Set

The 1M Movielens dataset [43] from the GroupLens Reseach Lab[2] is used to train and test the implemented deep reinforcement agents. Table 4.1 summarizes the statistical nature of the dataset. The ratings file was pre-processed in order to generate the user rating matrix, the transition probability matrix and to train and test the baseline CF-model using 5-fold cross validation to predict the missing ratings under the simulated environment.

| 1M Movielens dataset | |
|---|---|
| Total ratings | 1,000,029 |
| Users | 6,040 |
| Movies | 3,883 |
| Sparsity Level | 57.4% |
| Mean ratings/user | 165 |
| Min. ratings/user | 20 |
| Max. ratings/user | 2314 |

**Table 4.1:** Properties of the 1M Movielens dataset

[2] `http://grouplens.org/datasets/movielens/`

## 4.3 Evaluation Scheme

## 4.4 Baselines and Experimental Settings

To evaluate the strength and accuracy of deep reinforcement learning model, we compare the new model against a pure CF model and two variants of the baseline model described in section 3.2, The description of each model is listed as follows:

- **FM-MCMC**: *Factorization Machine with Monte Carlo Markov Chain inference* uses FM based on the Bayeasian inference with a Gibbs sampling technique that generates the distribution of $\hat{y}$ by sampling the conditional posterior distribution for each model parameter. This model was chosen as the baseline model for comparison as it it outperforms other learning approaches like SGD and ALS, and also because it integrates the regularization parameter into the model, which avoids a time-consuming search for the optimal hyperparameters. The only hyperparameter that remains for MCMC is the initialization of the standard deviation$\sigma$ for the Gibbs sampler.

- **DRL-kNN-CB**: *DRL with Content-based similarity for continuous action spaces* model uses the DDPG algorithm with a bag-of-words feature vectors to represent actions and states. The k-nearest neighbors index is then created using a KDtree algorithm[***] to find the closest set of existing items from the action estimated by the actor function. Finally, the Wolpertinger policy will apply the action in $\mathcal{A}$ that yields to the maximum return.

- **DRL-kNN-CF**: *DRL with Item-based similarity* model uses a similar configuration as the model described above but it implements an item-to item version of the k-NN algorithm for CF using the item vectors from the user rating matrix to find the closest set of items from an action given by the actor function.

- **DRL-FM**: *DRL with Factorization Machines policy* model replaces the k-NN algorithm from the models previously described with the FM-BRP model to return the Top-k items that can be recommended from a given state. Then, the FM-policy will apply the action from the top-N recommendations that yields to the highest Q value in the long term.

For the content-based model, feature vectors for each item were created using a combination of word embeddings extracted using *word2vec*[3], and genre categorization as a binary vector.

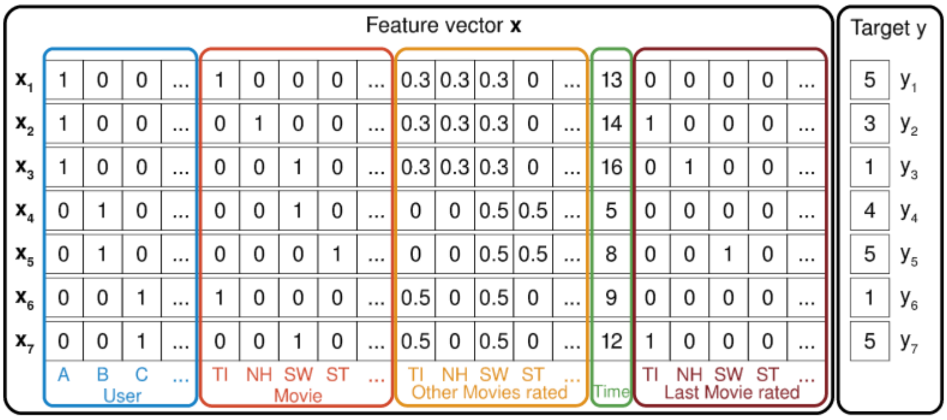
For the FM models, we use the *fastFM*[4] library introduced by Bayer I. in [44] that offers an efficient python implementationn of the FM learning algorithms including MCMC and BRP-OPT with SGDthe ratings file was transformed to the $SVM^{light}$ representation introduced in [45]. Figure 4.1 shows an example of the feature vector representation. First, there are $|\mathcal{U}|$ binary indicators variables (blue) that represent the active user of a transaction. The next $|\mathcal{I}|$ binary indicator variables (red) hold the active item. Each feature vector also contains normalized indicator variables (yellow) for all other items the user has ever rated. Finally, the vector contains a variable (green) holding the time when the rating was registered, and another variable containing information of the last movie the user has rated before.

In the experiments, (hyperparameter configuration for each model)[***Missing]

## 4.5 Results

[3] https://radimrehurek.com/gensim/index.html

[4] https://github.com/ibayer/fastFM



Feature vector $\mathbf{x}$

| | User | | | | Movie | | | | | Other Movies rated | | | | | Time | Last Movie rated | | | | | Target y | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | ... | TI | NH | SW | ST | ... | TI | NH | SW | ST | ... | | TI | NH | SW | ST | ... | | |
| $\mathbf{x}_1$ | 1 | 0 | 0 | ... | 1 | 0 | 0 | 0 | ... | 0.3 | 0.3 | 0.3 | 0 | ... | 13 | 0 | 0 | 0 | 0 | ... | 5 | $y_1$ |
| $\mathbf{x}_2$ | 1 | 0 | 0 | ... | 0 | 1 | 0 | 0 | ... | 0.3 | 0.3 | 0.3 | 0 | ... | 14 | 1 | 0 | 0 | 0 | ... | 3 | $y_2$ |
| $\mathbf{x}_3$ | 1 | 0 | 0 | ... | 0 | 0 | 1 | 0 | ... | 0.3 | 0.3 | 0.3 | 0 | ... | 16 | 0 | 1 | 0 | 0 | ... | 1 | $y_3$ |
| $\mathbf{x}_4$ | 0 | 1 | 0 | ... | 0 | 0 | 1 | 0 | ... | 0 | 0 | 0.5 | 0.5 | ... | 5 | 0 | 0 | 0 | 0 | ... | 4 | $y_4$ |
| $\mathbf{x}_5$ | 0 | 1 | 0 | ... | 0 | 0 | 0 | 1 | ... | 0 | 0 | 0.5 | 0.5 | ... | 8 | 0 | 0 | 1 | 0 | ... | 5 | $y_5$ |
| $\mathbf{x}_6$ | 0 | 0 | 1 | ... | 1 | 0 | 0 | 0 | ... | 0.5 | 0 | 0.5 | 0 | ... | 9 | 0 | 0 | 0 | 0 | ... | 1 | $y_6$ |
| $\mathbf{x}_7$ | 0 | 0 | 1 | ... | 0 | 0 | 1 | 0 | ... | 0.5 | 0 | 0.5 | 0 | ... | 12 | 1 | 0 | 0 | 0 | ... | 5 | $y_7$ |

**Figure 4.1:** Feature vector representation for ratings under the FM models (Source: Factorization machines[3])

## Chapter 5

# Conclusion and Discussion

## 5.1 Summary

## 5.2 Future Work

-More robust definition of the simulated environment

–robust model of user behaviour

–using a live recommender system

–monte carlo tree search

-Try to solve the problem using the Q-learning Model-based acceleration algorithm: Normalized Advantage Functions (NAF) to apply Q-learning with experience replay to continuous tasks. it outperforms actor-critic based algorithm (DDPG)

-test the model using dataset with higher sparsity level

## Chapter 6

# DDPG Algorithm using the FM-policy Architecture

# Appendix A

# Transition probability Matrix algorithm

**Algorithm 2:** Transition Probability Matrix(R)

```
Input: $R$: Initial User Rating Matrix
Result: $\mathcal{P}$: Transition Probability Matrix
$S \leftarrow ComputeCosineSimilarityMatrix(R)$
for $i \leftarrow 1$ to $m$ do
    $sum \leftarrow \sum_{j=1}^{m} S_{ij}$
    for $j \leftarrow 1$ to $m$ do
    /* user walks through a direct neighborith probability $\beta$
       or jumps to an arbitrary item with uniform
       probability                                          */
    $\mathcal{P}_{ij} \leftarrow \beta S_{ij}/sum + (1 - \beta)/m$
    end
end
return $\mathcal{P}$
```

# Appendix B

# Recommender System Simulated Environment

**Algorithm 3:** Simulated environment

```
Input: $R$: Ratings Matrix; $\mathcal{P}$: Transition Prob. Matrix; $\mathcal{I}$: Items set
/* selects an active user                                          */
$u_{random} \leftarrow selectRandomUser(R)$
/* initial guided exploration                                      */
$state \leftarrow selectLikelyItem(\mathcal{I}, u_{random})$
$term \leftarrow False$
while $not\ term$ do
    $action \leftarrow waitForAgentAction()$
    /* probability of choosing action given current state          */
    $\rho_{action} = \leftarrow computeTransition(\mathcal{P}, state, action)$
    $rating \leftarrow getUserRating(u_{random}, action)$
    /* get the mean and std dev of the transition
       probability distr.  for a given state                       */
    $\mu, \sigma \leftarrow transProbDistribution(\mathcal{P}, state)$
    $is\_chosen \leftarrow \rho_{action} < random.normal(\mu, \sigma) \&\& rating > 0$
    if $is\_chosen$ then
        $\rho_{end} \leftarrow 0.1$
        $reward \leftarrow rating$
    else
        $\rho_{end} \leftarrow 0.2$
        $reward \leftarrow 0$
        $action \leftarrow selectRandomItem(u_{random})$
    $state \leftarrow action$
    if $random.uniform() < \rho_{end}$ then
        $term \leftarrow True$
    $sendSignalToAgent(state, reward, term)$
```

# Bibliography

[1] Jesús Bobadilla, Fernando Ortega, Antonio Hernando, and Abraham Gutiérrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109–132, 2013.

[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[3] Steffen Rendle. Factorization machines. In *2010 IEEE International Conference on Data Mining*, pages 995–1000. IEEE, 2010.

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

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

[6] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1235–1244. ACM, 2015.

[7] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[8] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep Reinforcement Learning in Large Discrete Action Spaces. 2015.

[9] Prem Melville and Vikas Sindhwani. Recommender systems. In *Encyclopedia of machine learning*, pages 829–838. Springer, 2011.

[10] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186. ACM, 1994.

[11] Hilmi Yildirim and Mukkai S Krishnamoorthy. A random walk method for alleviating the sparsity problem in collaborative filtering. In *Proceedings of the 2008 ACM conference on Recommender systems*, pages 131–138. ACM, 2008.

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

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

[14] Yehuda Koren, Robert Bell, Chris Volinsky, et al. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[15] Marko Balabanović and Yoav Shoham. Fab: content-based, collaborative recommendation. *Communications of the ACM*, 40(3):66–72, 1997.

[16] Michael Pazzani and Daniel Billsus. Learning and revising user profiles: The identification of interesting web sites. *Machine learning*, 27(3):313–331, 1997.

[17] Prem Melville, Raymond J Mooney, and Ramadass Nagarajan. Content-boosted collaborative filtering for improved recommendations. In *Aaai/iaai*, pages 187–192, 2002.

[18] Steffen Rendle. Factorization machines with libfm. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 3(3):57, 2012.

[19] Steffen Rendle, Zeno Gantner, Christoph Freudenthaler, and Lars Schmidt-Thieme. Fast context-aware recommendations with factorization machines. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 635–644. ACM, 2011.

[20] Christoph Freudenthaler, Lars Schmidt-Thieme, and Steffen Rendle. Bayesian factorization machines. 2011.

[21] Yao Wu, Christopher DuBois, Alice X Zheng, and Martin Ester. Collaborative denoising auto-encoders for top-n recommender systems. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, pages 153–162. ACM, 2016.

[22] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[23] Michael N Katehakis and Arthur F Veinott Jr. The multi-armed bandit problem: decomposition and computation. *Mathematics of Operations Research*, 12(2):262–268, 1987.

[24] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[25] Stephan Ten Hagen, Maarten Van Someren, and Vera Hollink. Exploration/exploitation in adaptive recommender systems. *proceedings of Eunite 2003*, 2003.

[26] Pornthep Rojanavasu, Phaitoon Srinil, and Ouen Pinngern. New recommendation system using reinforcement learning. *Special Issue of the Intl. J. Computer, the Internet and Management*, 13, 2005.

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

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

[29] Daniela Godoy and Analía Amandi. Modeling user interests by conceptual clustering. *Information Systems*, 31(4):247–265, 2006.

[30] Elad Liebman, Maytal Saar-Tsechansky, and Peter Stone. Dj-mc: A reinforcement-learning agent for music playlist recommendation. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 591–599. International Foundation for Autonomous Agents and Multiagent Systems, 2015.

[31] Guillaume Chaslot. Monte-carlo tree search. *Maastricht: Universiteit Maastricht*, 2010.

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

[33] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[34] Sander Adam, Lucian Busoniu, and Robert Babuska. Experience replay for real-time reinforcement learning control. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(2):201–212, 2012.

[35] Marc G Bellemare, Joel Veness, and Michael Bowling. Investigating contingency awareness using atari 2600 games. In *AAAI*, 2012.

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

[37] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic Policy Gradient Algorithms. *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395, 2014.

[38] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[39] Enrico Bibbona, Gianna Panfilo, and Patrizia Tavella. The ornstein–uhlenbeck process as a model of a low pass filtered white noise. *Metrologia*, 45(6):S117, 2008.

[40] Andrew Zimdars, David Maxwell Chickering, and Christopher Meek. Using temporal data for making recommendations. In *Pro-*

*ceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pages 580–588. Morgan Kaufmann Publishers Inc., 2001.

[41] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. In *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence*, pages 452–461. AUAI Press, 2009.

[42] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[43] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4):19, 2016.

[44] Immanuel Bayer. Fastfm: a library for factorization machines. *arXiv preprint arXiv:1505.00641*, 2015.

[45] Thorsten Joachims. Making large scale svm learning practical. Technical report, Universität Dortmund, 1999.