Deep Reinforcement Learning for Dynamic Things of Interest Recommendation in Intelligent Ambient Environment

May S. Altulyan[1,3], Chaoran Huang[1], Lina Yao[1], Xianzhi Wang[2], and Salil Kanhere[1]

[1] The University of New South Wales, Sydney, NSW 2052, Australia
{m.altulyan, lina.yao, chaoran.huang, salil.kanhere}@unsw.edu.au
[2] The University of Technology Sydney, Ultimo, NSW 2007, Australia
xianzhi.wang@uts.edu.au
[3] Prince Sattam Bin Abdulaziz University, Alkharj 11942, Saudi Arabia
m.altulayan@psau.edu.sa

Abstract. Recommender Systems for the IoT (RSIoT) aim for interactive item recommendations. Most existing methods focus on user feedback and have limitations in dealing with dynamic environments. Deep Reinforcement Learning (DRL) can deal with dynamic environments and conduct updates without waiting for user feedback. In this study, we design a Reminder Care System (RCS) to harness the advantages of deep reinforcement learning in addressing two main issues of RSIoT: capturing dynamicity patterns of human activities and system update without a focus on user feedback. The RCS is formulated based on a Deep Q-Network (DQN), which works well with the dynamic nature of human activities. We further consider harvesting the feedback automatically in the back end without requiring users to explicitly label activities. Experiments are conducted on three public datasets and have demonstrated the performance of our proposed system.

Keywords: Deep Reinforcement Learning · IoT · Recommender System.

1 Introduction

With the rapid growth in the number of things that can be connected to the internet, Recommendation Systems for the IoT (RSIoT) have become more significant in helping a variety of applications to meet user preferences, and such applications can be smart home, smart tourism, smart parking, m-health and so on. On the one hand, RSIoT can recommend an item that users might need in situations. On the other hand, it can save time and cost by actively allocating specific IoT resources accordingly to the very situations.

We motivate RSIoT with a smart home scenario: Alice, a 79-year-old woman with dementia, lives alone in a house and is preparing a cup of coffee in her smart kitchen. Motion sensors monitor her every move and track each coffee-making

step. If she pauses for a while, a recommender application will determine if it is too long and remind her of what to do next. If she tries to prepare a cup of coffee late at night, the system considers the time and recommends she goes back to bed instead. Later that day, AliceâĂŹs son accesses the system and scans a checklist for his motherâĂŹs house. He finds that his mother has taken medicine on schedule, slept, eaten regularly, and continued to manage her daily life well.

Numerous efforts have been made to develop RSIoT using different approaches. Most of the existing works adapted conventional recommender system approaches, including collaborative filtering [?], content-based [?] and hybrid-based approach [?]. However, those conventional RSIoT approaches face two main issues. The first issue is treating the recommendation procedures as statics and ignoring the dynamicity in human activity patterns. More formally, human activity patterns could be changed at any time during the day or even after a period of time. The second issue is making recommendations for users while the system must wait for user feedback to update itself. While this may provide the system with accurate labels, it can have an impact on the end-user experience. RSIoTs should able to be updated based on the recommended item status only, which means no need to hold any device or to deal with any application.

Deep Reinforcement Learning (DRL) is inherently profitable for overcoming dynamic environments and thus has been adapted in interactive recommendation systems. It has been shown the ability to learn user decision behavior by observing the userâĂŹs actions and conducting accurate recommendations even from a few samples by grouping its observations. Furthermore, it considers the feedback from the environment as a reward to update the system. Significant efforts have shown the notable performance of DRL methods in conventional recommendation systems [?,?,?]. Also, there are only very few studies on RSIoT systems based on RL [?,?,?,?]. However, no previous research known to us has adapted DQN based RL for RSIoT. Inspired by [?], we design a Reminder Care System (RCS) based on DQN, which can tackle two main issues: dynamicity patterns of the human activity and the focus on the user feedback during system updates. We first formulate our system based on a Deep Q-Network (DQN), which captures the userâĂŹs dynamicity pattern using three kinds of extracted features that address the first issue. Subsequently, we calculate the probabilities for items and nominate only one item with the highest probability as a recommendation. To tackle the second issue, we introduce our reward function that enables the system to receive feedback automatically without waiting for the user. Finally, we propose a new term called a Reward Delay Period which improves the evaluation for the quality of recommendations.

The main contributions of our proposed system are summarized as follows:

- We design the Reminder Care System (RCS) and formulate it based on the Deep Q-Network (DQN) which utilizes three main features: past activities features, current activities features, and item context features as an input (State).

- We formulate the reward function that helps the system to be updated automatically without needing feedback from the user by checking the status of items after a period of time.
- We conducted extensive experiments on three public datasets, and our experimental results demonstrate the feasibility and effectiveness of our system.

2 Related work

2.1 Recommender system approaches for the IoT

Numerous recent works provide methods and techniques for building recommender systems in several domains of the IoT. Most of the existing research falls into three categories, the same as normal recommender systems: collaborative filtering, content-based, and hybrid methods. Authors in [?,?] proposed a unified CF model based on a probabilistic matrix factorization recommender system that exploits three kinds of relations to extract the latent factors among these relations. In [?], content-based was adapted for the recommender engine in their AGILE project, which aims to improve the health conditions of users. In [?,?], authors built their recommender system engine using a hybrid recommendation algorithm. All previous categories only focus on the interaction between items and users to construct a recommendation, making them inapplicable for RSIoT.

2.2 Deep Reinforcement Learning in RSIoT

In previous studies, most approaches deal with a static recommendation process, whereas RSIoT needs to capture user's temporal intentions and to conduct recommendations in a timely manner. DRL has received significant attention in building recommender systems [?,?,?,?,?] for two main reasons; coping with dynamic environments by updating the strategies during the interactions and the ability to learn a policy that maximizes the long term reward.

Author in [?] proposed a deep recommender system framework (DEERS). It aims to exploit both negative and positive feedback to conduct recommendations in a sequential interaction environment. In [?], DRL was adapted for the page-wise recommendation. The authors in [?] proposed a novel DRL-based recommendation framework. It tackles two main issues: the dynamic nature of new features and users' preferences and the lack of information to improve the quality of recommendations. In addition, DRL has also been utilized to propose a DEAR framework for online advertising recommendations[?].

3 Reminder Care System (RCS) framework

In this section, we introduce Reminder Care System (RCS) in detail. First, we define the problem and notations; then, we provide an overview of our framework. Finally, we describe DQN and explain the process of our agent.

3.1 Notation

Our problem is framed as follows: when extracted features of the complex activity v where $v_i \in V = \{v_1, v_2, ...v_m\}$ is received by the agent G (The extracted features will be explained in details in Section 3.2). Notice that the agent receives the extracted features of the activity that needs recommendation only as an input (state) s. Then the agent nominates an appropriate item a from a fixed candidate set of items A for the particular activity. In other words, the algorithm generates ranking list $\Gamma =< \gamma_{a_1}, \gamma_{a_2}, ..., \gamma_{a_l} >$, where γ_{a_i} denotes the probability of the item a_i where the user needs to finish the current activity. Unlike a conventional recommendation system that typically recommends more than one item for users each time, our agent recommends only one item with maximum probability for the activity that needs a recommendation. Table 1 summarises the notations used throughout this paper.

Table 1: System Notations

Notation	Explanation
G	Agent
v, V	Activity, set of activities
s, s'	state, next state
a	action(item)
r	Reward
A	set of items
Q	Q-Network
W	Parameters of DQN parameter
Γ	Ranking list
γ_a	Probability of the item
O	Value of each item
E	Experience replay buffer

3.2 Overview

In this section, we describe our framework as shown in Fig.1. We divided it into two main parts online and offline. In the offline part, our system will be training to deal with the activities that need a recommendation. Notice during the training, we treat each activity that needs a recommendation as a session. During the online part, the agent receives the required features as an input (state) s; then recommends an appropriate item a_i for the activity. There are three kinds of features that should be received for each activity (session) that needs a recommendation:

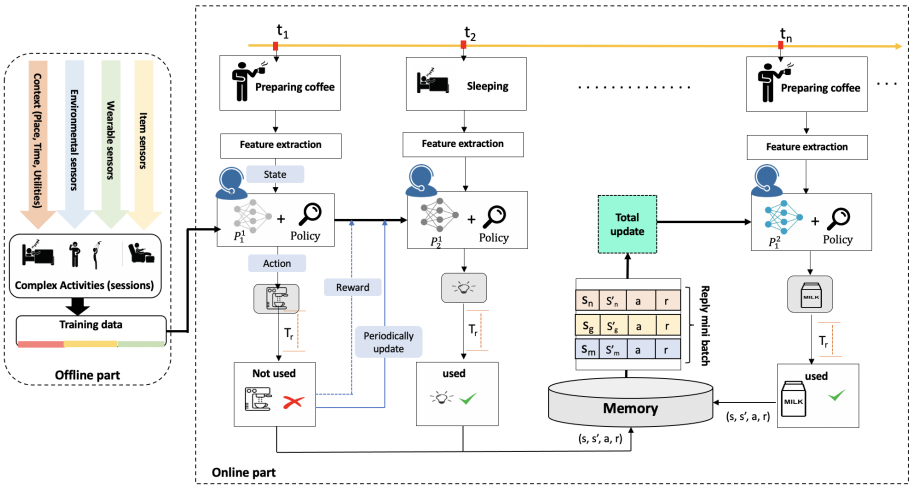


Fig. 1: The architecture of our framework, which consists of two main parts: offline and online. The offline part focuses on training our agent G using different datasets. During the online part, the agent receives each activity and extracts the required features as state s, then nominates suitable item a_i for this state. After a periodic time called reward delay period T_r, the agent will receive a reward r. Two kinds of updating will be applied for the system: periodically update after every recommendation P and the total update after a period of time using the experience replay buffer.

- Past activities features. Since each activity can have a different pattern, for each activity, the system extracts the path/sequences of items used. They enable agent G to learn different patterns of each activity.
- Current activities features. It can define where the user is stuck by reviewing all previously used items for this activity. This feature helps the agent to ignore all the used items before and to choose from the rest.
- Item context (IC). It includes information about items, such as to which activity this item belongs to, how long it could be in use, and how many times the user needs it for the current activity.

  To improve the system's recommendation accuracy, we consider all features that help our system learn the best action in a specific state. The public datasets did not meet all required features , which affect the results of our systems.

3.3 Deep Q network for recommendation

After the features are extracted, we apply the DQN algorithm to model our agent. It maps state and action pairs to Q-value using neural networks. It aims to maximize a cumulative future reward by recommending the correct item a_i

for each activity that needs a recommendation. Three main components for the DQN are Q-Network which could be a standard neural network or regular network depending on the state; Q-Target is identical to the Q Network; and Experience Replay, which stores all the interactions with the environment and uses them as mini-batch to update the Q-network. The DQN has two main features compared with other RL algorithms: (1) using the experience replay buffer to store the agent experiences $E = \left\{ s_i, a_i, r_i, s_i^{'} \right\}$ which represents state, action, reward, and next state respectively, (2) adjusting any update for the target network. Here, agent G (Q-network) will be trained using the offline part. During this part, the agent learns to map each stat for suitable action. Then, the agent calculates the reward as feedback to updated the system. We summary the agent roles as following:

1. Receiving the extracted features of current state s during the interaction with the environment at timestamp t.
2. Generating a list of recommendations Γ that includes top items to be recommended using exploration and exploitation policy. Notice, the agent will pick only one item with the highest ranking.
3. Calculating the reward which considers as feedback to update the system using the following equation:

$$r(a) = \frac{\sum_{t=0}^{T_r} O_{a,t}}{\sum_{a=0}^{A} \sum_{t=0}^{T_r} O_{a,t}} \tag{1}$$

   where O represents the value of each item at each time step and T_r is a Reward Delay Period.
   However, the agent G has to wait for the reward delay period T_r (will be discussed in the next part).
4. Periodically updating after every recommendation by comparing the performance of the Q-network with target network using the following loss fucntion:

$$loss = MES(predicted\ Q - Value, Target\ Q - Value) \tag{2}$$

5. Total updating for the system to tackle the dynamicity of human activity pattern, the agent G after a period of time (it is defined to be 24 hours for our system) will use the experience replay buffer to update the network Q using the following loss equation [**?**]:

$$L_i(\theta_i) = E_{(s,a,r,s')} \sim U(D) \left[\left(r + \gamma max_{a'} Q(s^{'}, a^{'}; \theta_i^{-}) - Q(s, a; \theta_i) \right)^2 \right] \tag{3}$$

   where γ is a discount factor , θ_i and θ_i^{-} represent the parameters of the Q-network and the target-network at iteration i respectively.

Most traditional recommender systems focus on "click" or "not click" as feedback to calculate the reward function immediately and to update the system. In

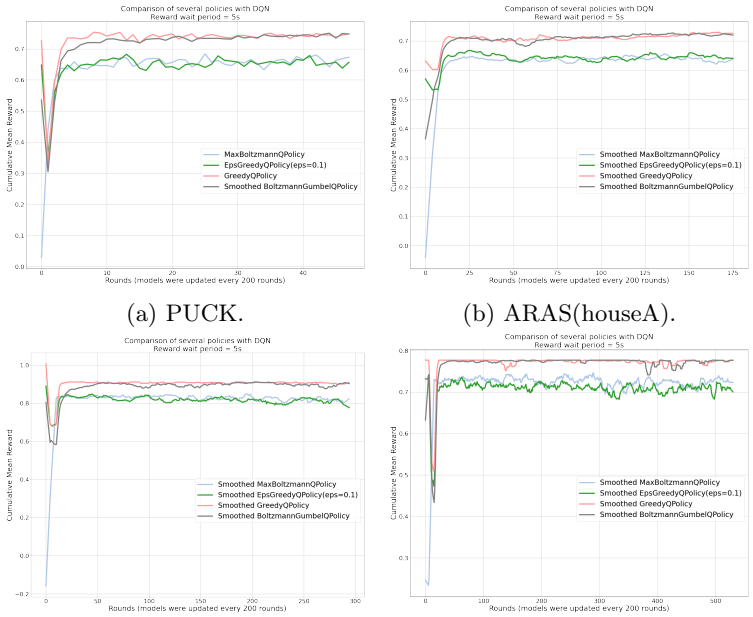
contrast, our system recommends an item to the user and then waits for sufficient time to decide if the recommended item is used or notâĂŤby checking its status (on/off or moved/not moved). For example, if the system recommends a coffee machine to Alice when she is preparing a cup of coffee, whereas she wants to use it later yet not immediately. This does not mean that the recommended item is incorrect, and it is better for the system to ignore this false negative feedback this time. To facilitate the above, we introduce a Reward Delay Period T_r, which accounts for the different paces of users in carrying out activities, and we consider T_r a hyperparameter. Here, our agent acts using different policies: EpsGreedyQPolicy, GreedyQPolicy, BoltzmannGumbelQPolicy, MaxBoltzmannQPolicy. Table 2 shows details about the policies with DQN that we used for our agent and their parameters.

Table 2: Explaining policies and tuning Hyperparameters for our agent.

Policies	
(1) EpsGreedyQPolicy:	it combines both exploration by taking a random action with probability epsilon and exploitation by taking the current best action with probability (1 - epsilon)
(2) GreedyQPolicy:	it focuses on exploration where it calculates the probability of choosing the action with the highest Q-value
(3) BoltzmannGumbelQPolicy:	it is an exploration rule which defines probabilities of actions based on their Q-values.
(4) MaxBoltzmannQPolicy:	it adapts the Gumbel-softmax trick to address the classic Boltzmann exploration issues
Hyperparameters	
batch_size	200
Epsilon	0.1
target_model_update	1e-3
nb_steps	50000
verbose	1

4 Experiment

In this section, we first introduce three public datasets that we used for our experiments. Next, we conduct some experiments that show the effectiveness of our proposed RCS and evaluate the performance among these datasets.



(a) PUCK. (b) ARAS(houseA).

(c) ARAS(houseB). (d) ADL.

Fig. 2: Performance of our system among three datasets and the T_r=5.

4.1 Datasets

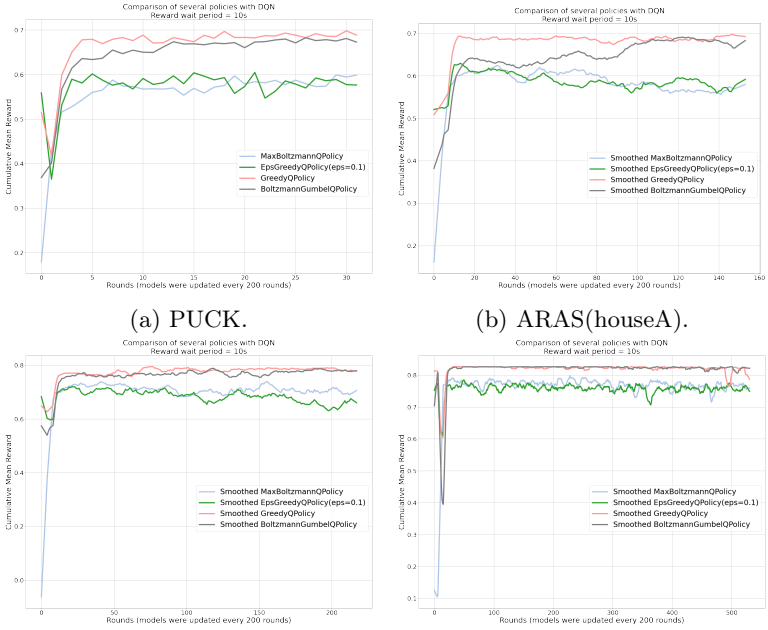
Our evaluation focused on the offline part, and we left the evaluation of the online part to future work. The evaluation has been applied on three public datasets: PUCK [**?**], ARAS [**?**], and ADL [**?**].

The PUCK dataset[4] was collected in Kyoto smart home testbed in Washington State University, and it consists of a two-story apartment with one living room, one dining area, one kitchen, one bathroom, and three bedrooms. A number of environmental sensors have been installed in this testbed. The ARAS dataset contains two houses of two residents who performed 27 daily living activities. The activity-sensory data was collected from 20 binary sensors. The ADL Normal dataset represents a public dataset published in 2010. The dataset was collected from a Kyoto smart apartment testbed in Washington State University. It includes five complex activities. The activities are performed by 20 participants. Features engineering and data processed are explained in detail in our previous work [**?**].

4.2 Experiments results

We first evaluate the effectiveness of our RCS in recommending the correct item to the user in case the user's current activity needs a recommendation. The

[4] http://casas.wsu.edu/datasets/puck.zip



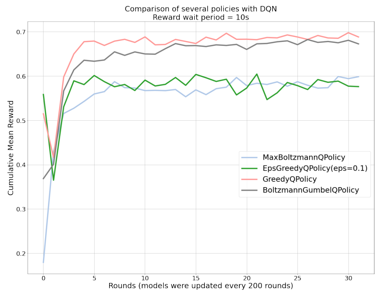
(a) PUCK. (b) ARAS(houseA).

(c) ARAS(houseB). (d) ADL.

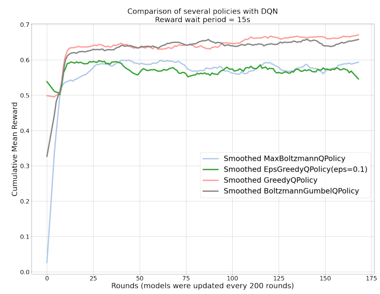
Fig. 3: The Reward Delay Periods T_r=10s.

agent uses all the extracted features as the state to make a recommendation of the correct item. We use the DQN model that is provided by one public available 'keras-rl' package of python for our experiments. The hyper-parameters of the DQN model are configured as follows: the number of layers in DQN is set to 6 with two Flatten layers, three Dense layers, and one Activation layer. The package provides a number of policies that help our agent to map each state with a correct action.

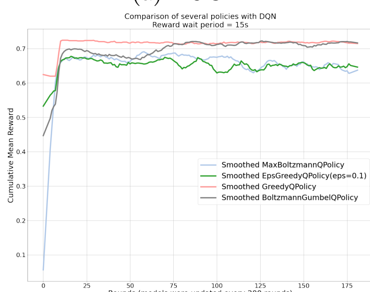
The performance of our system is shown in Fig. 2. As we can see, the cumulative mean reward for the three datasets; However, the two policies: GreedyQpolicy and BlotzmanngumbleQpolicy, produce the highest performance compared with another two policies. Also, the ARAS dataset for house B (see Fig. 2c) has the highest cumulative mean reward compared with other datasets. There are different reasons that could affect the results on different datasets: (1) the number of item sensors used to collect the data, (2) the time period between each reading of the sensors values, i.e., half a second for PUCK and a second for ARAS and ADL datasets, (3) the type of sensor value such as binary or continues values, (4) the number of items that are included in each activity. Moreover, all the previous datasets do not consider time as a context which is an important feature for our system. As mentioned, the Reward Delay Period parameter T_r has direct impacts on the model's performance which controls when the agent should receive the feedback as a reward. Adjusting this parameter is important,



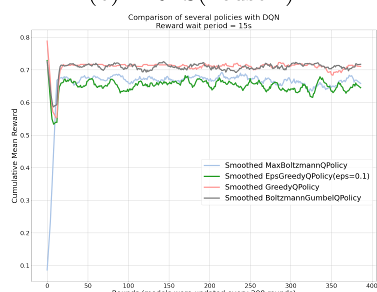
(a) PUCK.



(b) ARAS(houseA).



(c) ARAS(houseB).



(d) ADL.

Fig. 4: The Reward Delay Periods T_r=15s.

Table 3: The cumulative mean reward of our system among three datasets using different reward delay periods.

Dataset	T_{r}	Policies			
		EpsGreedyQPolicy	GreedyQPolicy	BoltzmannGumbleQPolicy	MaxBoltzmannQPolicy
PUCK	5s	0.65	**0.75**	**0.75**	0.67
	10s	0.58	**0.68**	0.66	0.60
	15s	0.51	**0.64**	0.62	0.52
ARAS House (A)	5s	0.64	**0.72**	0.71	0.63
	10s	0.59	**0.69**	0.68	0.57
	15s	0.54	**0.67**	0.66	0.59
ARAS House (B)	5s	0.77	**0.90**	**0.90**	0.83
	10s	0.68	**0.78**	**0.78**	0.70
	15s	0.65	**0.73**	**0.73**	0.64
ADL	5s	0.70	**0.77**	**0.77**	0.72
	10s	0.74	0.77	**0.82**	0.75
	15s	0.64	0.71	**0.72**	0.65

and it is various from one activity to another depending on how much time each item consumes to be used. For example, some items take a little bit of time to be picked, and others may be a little bit longer. We assume three values of T_r: 5s, 10s, and 15s, then we monitor the performance among these different values. Fig. 2, Fig. 3 and, Fig. 4 show the performance of our RCS on the three datasets using three different values of T_r: 5s, 10s, and 15s respectively. Table 3 sum-

marizes all cumulative mean rewards of our system among three datasets using different reward delay periods. We can observe that our proposed system performance when the T_r=5 consistently outperforms the other two values: T_r=10, and T_r=15 for the two datasets: PUCK, and ARAS. However, the ADL dataset demonstrates that increasing the T_r to 10s improves the cumulative reward to be around 0.82 instead of 0.77 and 0.72 for the T_r=5 and T_r=15 respectively. Moreover, table 3 shows the effectiveness of The GreedyQPolicy and BoltzmannGumbleQPolicy policies compared with the other two policies. However, among all the policies, the GreedyQPolicy performs well and is stable except in the last dataset with T_r=10.

5 Conclusion

In this study, we designed a Reminder Care System (RCS) that uses deep reinforcement learning to capture dynamic patterns of human activities and update the system automatically without waiting for user feedback. The RCS uses DQN to formulate the agent and considers the reward delay period to account for the different paces of users in carrying out activities. We conducted experiments on three real-world public datasets to show that the effectiveness of our system. For future work, we will test our system on a real-time testbed that considers all features requirements for the proposed system.