Network Friendly Recommendation System

Ioannis Kontogiorgakis[1]

[a]*ECE, Technical University of Crete,*

Abstract

The rapid growth of online content consumption has led to an overwhelming abundance of choices for users. Recommendation systems play a crucial role in assisting users in discovering relevant and engaging content tailored to their preferences. However, the increasing demand for personalized recommendations and the need to accommodate network constraints present significant challenges for efficient content delivery. In this project, we aim to develop a Network Friendly Recommendation System that addresses these challenges by integrating principles from reinforcement learning and network optimization. Our objective is to create an intelligent system that optimizes content recommendations while minimizing the impact on network resources. By modeling the recommendation problem as a Markov Decision Process (MDP), we can analyze and optimize the system's behavior by considering user preferences, network constraints, and content relevance.

Keywords: Reinforcement Learning, Recommendation System, Policy Iteration, Q-Learning, Deep Q-Networks

1. Introduction

The digital era has witnessed an exponential growth in online content consumption, offering users an immense variety of choices. With such vast options available, users often rely on recommendation systems to guide their content discovery process. These systems have become essential in providing personalized recommendations tailored to individual preferences. However, the increasing demand for personalization, combined with network constraints, presents unique challenges in delivering content efficiently.

The objective of this project is to develop a Network Friendly Recommendation System that tackles the aforementioned challenges. Our focus is to optimize the content recommendation process while considering both content relevance and network limitations. By integrating principles from reinforcement learning and network optimization, we aim to create an intelligent system capable of providing efficient recommendations while minimizing the impact on network resources.

To address these challenges, we model the recommendation problem as a Markov Decision Process (MDP). In this framework, we characterize the content items and user interactions as states, actions, and rewards. By utilizing the MDP framework, we can systematically analyze and optimize the behavior of the recommendation system. This approach allows us to incorporate the dynamics of user preferences, network constraints, and content relevance, enabling us to make informed decisions in the recommendation process.

By developing a Network Friendly Recommendation System, we aim to enhance the overall user experience by providing personalized and relevant content recommendations while considering the limitations imposed by network resources. Through the integration of reinforcement learning and network optimization techniques, we seek to create a robust and efficient recommendation system that adapts to individual user preferences and maximizes the utilization of available network resources. We will attempt to model the system's decision making using 3 Reinforcement Learning algorithms, Policy Iteration, Q-Learning and Deep Q-Network and we will analyze their interaction with respect to the system and the user.

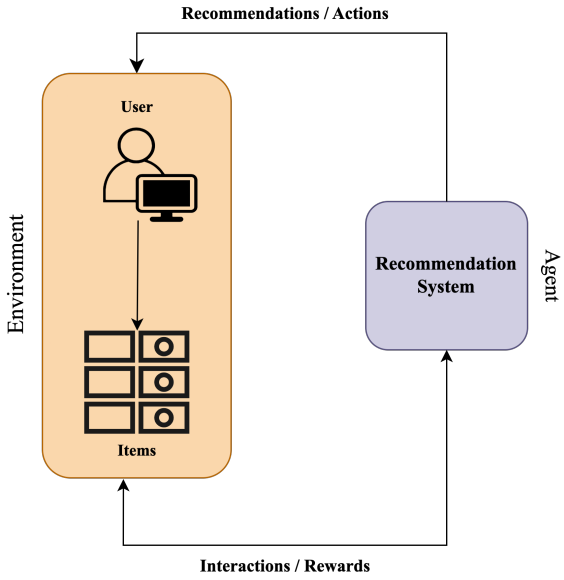

Figure 1: Environment overview.

2. Environment

The Network Friendly Recommendation System focuses on a content catalogue consisting of K items, each with varying degrees of relevance to one another. We explore the relationship between content items using a matrix that quantifies their relevance scores, the relevance matrix U. The relevance matrix of shape (K, K) is randomly generated from the normal distribution and each value U_{ij}, represent the relevance of item i with item j. Additionally, we introduce a threshold value to distinguish relevant and irrelevant content pairs. Furthermore, a subset of the content items is designated as cached items, which incur zero cost, while non-cached items have a cost associated with them. The User Model component of the system simulates user behavior by allowing them to watch multiple items during a session. After viewing a content item, the system recommends N new items to the user. The user's decision to continue watching or end the session is probabilistic, with considerations given to the relevance of the recommended items and the probability of selecting a recommended item versus choosing from the entire catalogue.

In order to construct our MDP, we employ the following environment:

- **States** : The state space consists of the current item the user is watching. Therefore, the states, denoted as s, range from 0 to K, representing the current item being watched by the user.
- **Actions** : The action space represents the N items to recommend to the user after they finish watching the current content item. For the case of $N = 2$, the action space is defined as tuples of size N from the remaining uncached items in the content catalog. Each tuple, denoted as $(item_1, item_2)$, is selected from the set $\{0, 1, ..., K\}$, where $item_1$ and $item_2$ are distinct items.
- **Transition Probabilities** : The transition probabilities in the MDP depend on the user's behavior. There are two scenarios: ending the session or continuing the session. With a probability of q, the user ends the session. Otherwise, with a probability of $1 - q$, the user continues the session. If both recommended items are relevant to the current item the user is watching, there is a probability of a to pick one of the N recommended items. Otherwise, with a probability of $1 - a$, the user picks one of the remaining items.
- **Rewards** : The reward in the MDP is defined based on whether the selected item is cached or not. If the selected item is cached, the reward is 0. If the selected item is not cached, the reward is -1. The negative reward encourages the system to recommend cached items to minimize the user's total cost.

An environment overview is summarized in Fig. 1

3. Algorithms

To solve the aforementioned MDP, we employ classical exact reinforcement learning methods : *Policy Iteration* and *Q-learning*, along with approximation methods (*Deep Q-Networks*). The algorithm iteratively improves the recommendation policy by evaluating the expected total cost of items watched during a session and updating the policy based on the evaluation results. Through training, the system learns to recommend the most relevant and network-friendly content items to users, resulting in an enhanced user experience and efficient network resource utilization. In the next subsections, we will delve into details for the implementation of each algorithm.

3.1. Policy Iteration

Policy Iteration is an algorithmic approach used to solve Markov Decision Processes (MDPs) and find an optimal policy. It involves two main steps: policy evaluation and policy improvement. Through iterations of these steps, the algorithm converges to an optimal policy that maximizes the expected return.

3.1.1. Policy Evaluation

In the policy evaluation step, we aim to determine the value function for a given policy. The value function represents the expected return from being in a particular state and following a specific policy. It is denoted as $V(s)$, where s is the state.

The value function satisfies the Bellman equation, which is defined as follows:

$$V(s) = \sum_{s' \in S} P_{ss'}^{\pi(s)} (R_{ss'} + \gamma \cdot V(s'))$$

where:

- $R_{ss'}$ is the immediate reward obtained by moving from state s in state s'.
- γ is the discount factor, which represents the importance of future rewards relative to immediate rewards.
- $P_{ss'}^{\pi(s)}$ is the transition probability of moving from state s to state s' under policy $\pi(s)$.
- $V(s')$ is the value function of the next state s'. The policy evaluation step involves iteratively updating the value function using the Bellman equation until it converges. This ensures that the value function accurately reflects the expected return under the given policy.

3.1.2. Policy Improvement

After obtaining the value function through policy evaluation, we proceed to the policy improvement step. In this step, we aim to improve the current policy by greedily selecting actions that lead to higher value states. The policy

improvement is based on the policy improvement theorem, which states that for any given policy π, if we construct a new policy π' that is greedy with respect to the value function $V(s)$ obtained from π, then the new policy π' will be equal to or better than the old policy π. The policy improvement step involves examining each state and its possible actions. We update the policy by selecting the action that maximizes the expected return according to the current value function $V(s)$. This process is repeated until the policy no longer changes, indicating that we have reached the optimal policy.

3.1.3. Convergence and Optimal Policy

Policy Iteration guarantees convergence to an optimal policy. The algorithm alternates between policy evaluation and policy improvement steps until the policy no longer changes during the improvement step. At this point, we have obtained the optimal policy that maximizes the expected return for each state. By applying Policy Iteration to the Markov Decision Process of the Network Friendly Recommendation System, we can iteratively evaluate and improve the policy until we converge to the optimal policy. This optimal policy will guide the recommendation system in selecting the most appropriate content items for the users, taking into account the user's behavior and minimizing their total cost.

3.2. Q-Learning

Q-Learning is a popular algorithm for solving reinforcement learning problems in the absence of a known model of the environment. It is a model-free, off-policy algorithm that learns an action-value function, denoted as $Q(s, a)$, which represents the expected cumulative reward for taking action a in state s and following a specific policy.

The Q-Learning algorithm updates the Q-values based on the observed rewards and transitions without requiring explicit knowledge of the transition probabilities. It gradually improves the action selection by iteratively updating the Q-values until convergence to the optimal action-value function.

The Q-value for a state-action pair (s, a) is updated using the following equation:

$$Q(s,a) \leftarrow (1-a) \cdot Q(s,a) + \alpha \cdot \big(R(s,a) + \gamma \cdot \max_{a'}(s',a')\big)$$

where:

- $Q(s, a)$ is the Q-value for the state-action pair (s, a).
- a is the learning rate, controlling the weight of the new information relative to the existing Q-value.
- $R(s, a)$ is the immediate reward obtained in state s, after action a.
- γ is the discount factor, representing the importance of future rewards.
- s' is the next state after taking action a in state s.
- a' is the action selected at the next state s'.

The Q-Learning algorithm iteratively, through a number of episodes, updates the Q-values, exploring different action selections, and gradually converges to the optimal action-value function. By repeatedly interacting with the environment, observing state transitions and rewards, the algorithm learns the best recommendations to minimize the total cost for users in the Network Friendly Recommendation System.

Through the Q-Learning process, we can derive an optimal action-value function that guides the recommendation process. The learned Q-values enable us to select the best actions, ensuring efficient content recommendations while considering network friendliness and minimizing user costs.

3.3. Deep Q-Network (DQN)

Deep Q-Networks (DQNs) have emerged as a pivotal advancement in the field of reinforcement learning due to their ability to tackle complex problems that were beyond the reach of traditional algorithms like Policy Iteration and Q-learning. These earlier algorithms struggled when confronted with large-scale environments, such as those involving a substantial number of items, such as $K = 1000$ items. Running experiments of this magnitude with traditional methods would take an impractical amount of time and resources. DQN addressed this challenge by leveraging deep neural networks to approximate the Q-values for state-action pairs.

DQN implementations typically consist of several key components: a neural network, experience replay, and optimization.

3.3.1. Neural Network - Policy Network

The neural network of this implementation, often referred as Policy Network employed in this project is defined as follows. The neural network comprises three layers, each serving a specific purpose:

1. **Input Layer:** The first layer, `layer1`, is a linear layer that accepts an input tensor of length $K + 1$. This input consists of the current state and the relevance scores of the current state in relation to the other items in the catalog.
2. **Hidden Layer:** The second layer, `layer2`, is another linear layer that takes the output of `layer1` as input. It has a hidden dimension of 128, meaning it has 128 neurons. This hidden layer applies a sigmoid activation function to its inputs, introducing non-linearity into the network.
3. **Output Layer:** The final layer, `layer3`, is also a linear layer responsible for producing the network's output. It has an output dimension of `action_dim`, which represents the number of possible actions or recommendations. The output tensor from this layer consists of K Q-values, each corresponding to a particular state-action pair.

In essence, this neural network processes information about the current state and its relevance to other items in the catalog, transforming it through multiple layers to produce a set of Q-values. These Q-values guide the decision-making process in reinforcement learning, indicating the estimated value of taking different actions in the current state. The use of hidden layers with non-linear activation functions allows the network to capture complex relationships within the input data.

3.3.2. Deep Q-Network Training

In order to tackle the problem of a large and discrete action space, we tried a different approach. In contrast to the previous algorithms, where action selection involved computing Q-values for all possible actions, our DQN takes a different approach : Q-values are computed for K states, each representing a recommendation, which is why the neural network has an output of shape K. During action selection in each episode, the algorithm chooses the N actions with the highest Q-values from the neural network's output. Mathematically, this can be expressed as:

$$\text{Selected Action} = \text{argmax}_N(\text{policy\_net}(\text{current\_state}))$$

Experience replay involves storing and randomly sampling past experiences, which helps stabilize training by breaking the temporal correlation of consecutive experiences. Optimization in DQN involves updating the policy network's weights to minimize the temporal difference error, based on the Bellman equation.

$$Q(s,a) = R(s,a) + \gamma \cdot \max_{a'}(s', a')$$

The difference error δ can be expressed as :

$$\delta = Q(s,a) - (R(s,a) + \gamma \cdot \max_{a'}(s', a'))$$

To minimize this error, we employ the Huber loss, a versatile loss function. The Huber loss resembles the mean squared error for small errors but behaves like the mean absolute error for larger errors. This adaptive behavior enhances robustness against outliers, especially when the Q-value estimates are highly noisy. We compute this loss across a batch of transitions, denoted as B, sampled from the replay memory.

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,s',r) \in B} \mathcal{L}(\delta)$$

$$\text{where} \quad \mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

As mentioned earlier, in order to tackle the problem of the large action space, we modified the neural network output in order to produce a Q-values of shape K, where each value represents the Q-value of a given state-recommendation and then choosing the best action by selecting the indices of the N max Q-values. For that reason and in order to perform the differentiation, we kept the mean of the N max Q-values pair for each state. Optimization process includes an important target network, which lags behind the policy network and aids in providing more stable target values. The weights of the target network are softly through an interpolation process, represented mathematically as:

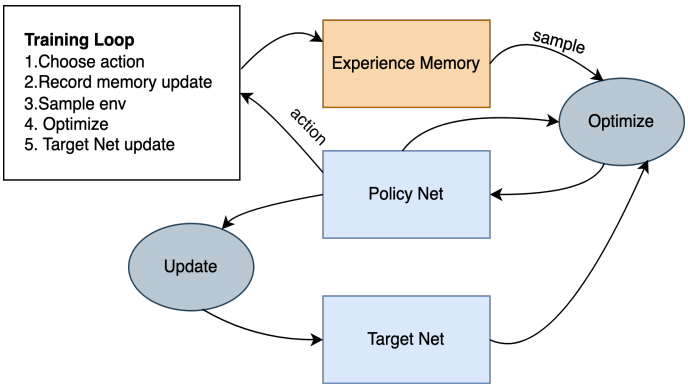

Figure 2: Architecture of our DQN : First, we utilize Policy Net in order to select a random or the max value action, by selecting the arg max N Q-values. After performing a step into our environment, we store the output into the experience memory buffer. Whenever buffer fulls, we perform optimization, by sampling experiences from our buffer. After each, episode, Target is softly updated according to Policy Net weights.

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$

where θ' are the weights of the target network, θ are the weights of the policy network, τ is the interpolation parameter, often referred to as the *soft update rate*. It controls how much of the policy network's weights are blended into the target network's weights. This soft update operation gradually updates the target network's weights to be more similar to the policy network's weights, which is essential for stabilizing and improving the training process in the Deep Q-Network. The architecture of the implementation is summarized in Figure 2.

4. Evaluation

In order to evaluate and compare the performance of the aforementioned algorithms, we designed and tested several scenarios - experiments, that reflect to simple real life recommendation system problems. We will use Policy Iteration as a benchmark for the optimal policy and we will compare the performance of Q-Learning and Deep Q-Networks in various scenarios. For this purpose, we will look into the average reward and cost gained through the episodes, and time of execution. To better visually understand the the extracted policy for each algorithm, we designed a policy heatmap according to relevance scores

and the cached hit of the recommendations. The heatmap is basically a vector of length K that summarizes the performance of the policy in a given state.The color values explained as follows:

#relevant items $= N$, #cached items $= N$.

#relevant items $= N$, #cached items $\geq N/2$.

#relevant items $= N$, #cached items $< N/2$.

#relevant items $< N$.

Figure 3: Color legend for policy heatmap

First we will delve into the experiments that involve tabular methods, Policy Iteration and Q-learning, followed up with several scenarios for Deep Q-Network evaluation.

4.1. Experiment: Convergence to Policy in a Toy scenario, Policy Iteration and Q-Learning

In order to evaluate our models performance, we created a toy scenario to demonstrate the convergence of our algorithms to an optimal policy. The objective of experiment is to model a small recommendation system with a small number of items, specifically $K = 10$, while keeping other parameters as specified below.

- **Number of Items** (K): We chose a small number of items to facilitate the understanding of the algorithm and the analysis of results.
- **Discount Factor**($gamma$): We set the discount factor (Gamma) to 0.99. This indicates that the system is focused on maximizing long-term rewards and encourages the user to continue watching and receiving recommendations.
- **User's Session Continuation Probability** (q): To simulate a user who is keen on continuing the session, we set the probability of session continuation (q) to 0.2. This means that with a probability of 0.2, the user will choose to watch another video after finishing the current one.
- **Probability of Choosing Recommended Items** (a): We assumed that the user is more likely to choose a recommended item. Therefore, we set the probability of choosing a recommended item (a) to 0.8. This encourages the system to focus on providing relevant recommendations to the user.
- **Relevance Threshold** (u_{min}): To ensure the neutrality of the system and in order to model an average user, we set the relevance threshold (u_{min}) to 0.3. This means that items with a relevance value below 0.3 are considered irrelevant and should not be recommended to the user, unless they are cached.

By setting up this toy scenario, we aimed to observe the convergence of the Policy Iteration and Q-Learning algorithm as it optimizes the policy towards recommending relevant and cached items to the user. Throughout the iterations, we expected to see the system gradually improve its recommendations and converge to an optimal policy that minimizes the cost and maximizes the user's satisfaction. The cached item in this scenario are **5** and **6**. The output policies for each algorithm, alongside with the output policy heatmap, are summarized in Fig. 4 and Fig. 5, for Policy Iteration and Q-Learning respectively.

In the context of the Policy Iteration approach, our recommender system has yielded noteworthy results. From the policy table in Fig. 4, we observe that the cached items **5** and **6** are mostly recommended to the user, highlighting the effectiveness of the system to reduce the total cost. Furthemore, the policy heatmap emphasize the ability of the system to produce relevant pair of recommendations, given the fact that if all N recommended items are relevant, the user will choose one of the recommendations with probability α. Overall, in the majority of the states all recommended items are relevant with respect to the current watching item(state), and all recommendations are cached. Someone can observe that states $4, 5$ and 6 suggest only one recommended cached item, and that is due to the the relevance score of the cached item, towards the current watching item. Precisely, in states 4 and 7, the first suggested item is 5 and 6 respectively, while the second item is 9 for both states. In this example, state(item) 4 is not relevant to cached item 6 and item(state) 7 is not relevant to cached item 5. States $5, 6$ necessarily suggest only one cached item,item 6 and item 5 respectively, since we restrained the recommendation system not to suggest the same item the user is currently watching.

The Q-Learning recommender policy, depicted in the table of the Fig. 5, appears to recommend fewer cached and relevant items compared to the Policy Iteration policy. In the Q-Learning approach, the recommendations seem to vary more widely in terms of relevance scores, often leaning towards higher values for the second item in the pair. This suggests that the Q-Learning policy may prioritize items based on their individual relevance rather than considering the caching aspect as heavily as the Policy Iteration policy. On the other hand, the Policy Iteration policy consistently emphasizes recommending cached items, as evident by the prevalence of the same action $(5, 6)$ across different states. This deliberate emphasis on cached items indicates an approach that optimizes for efficiency and speed in content delivery while maintaining a relatively high level of relevance, which is advantageous in real-time or time-sensitive scenarios. Ultimately, the choice between these policies would depend on the specific priorities of the recommendation system, whether it be efficiency, relevance, or a balanced combination of both.

	Policy Iteration		
State	**Item 1**	**Item 2**	**Relevance**
0	5	6	(0.3891, 0.6015)
1	5	6	(0.3863, 0.3725)
2	5	6	(0.6990, 0.4507)
3	5	6	(0.5973, 0.4493)
4	5	9	(0.3557, 0.5894)
5	6	9	(0.6675, 0.6293)
6	5	9	(0.6675, 0.5094)
7	6	9	(0.5947, 0.5254)
8	5	6	(0.7067, 0.3616)
9	5	6	(0.6293, 0.5094)

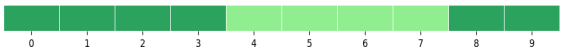

Figure 4: Top : Policy table extracted from Policy Iteration in the Toy scenario. Cached items are highlighted with light gray color. Bottom : Policy Heatmap of the extracted policy in the Toy scenario.

	Q-Learning		
State	**Item 1**	**Item 2**	**Relevance**
0	5	6	(0.3891, 0.6015)
1	5	6	(0.3863, 0.3725)
2	5	6	(0.6990, 0.4507)
3	0	5	(0.4115, 0.5973)
4	3	7	(0.1860, 0.4106)
5	2	6	(0.6990, 0.6675)
6	5	9	(0.6675, 0.5094)
7	5	8	(0.1153, 0.5201)
8	0	5	(0.4921, 0.7067)
9	5	6	(0.6293, 0.5094)

Figure 5: Top : Policy table extracted from Q-Learning in the Toy scenario. Cached items are highlighted with light gray color. Bottom : Policy Heatmap of the extracted policy in the Toy scenario.

4.2. Experiment: Convergence to Policy in a Toy scenario, Deep Q-Network

In this experiment, we investigate the convergence of the DQN algorithm as applied to a toy scenario. In this scenario, we consider a recommendation system with $K = 10$ items and aim to generate $N = 2$ recommendations. The goal is to examine how the algorithm, which we have implemented, adapts and learns to make optimal recommendations over time. For this experiment we used the same default environment and user configurations, in a 1000-episodes execution.

The Fig. 6 showcases the optimal policy derived from the Deep Q-Network (DQN) algorithm. Observing the policy table and comparing with the theoretically optimal policy

	Deep Q-Network		
State	**Item 1**	**Item 2**	**Relevance**
0	5	6	(0.3891, 0.6015)
1	5	6	(0.3863, 0.3725)
2	5	6	(0.6990, 0.4507)
3	5	6	(0.5973, 0.4493)
4	5	6	(0.3557, 0.2890)
5	3	6	(0.5973, 0.6675)
6	3	5	(0.4493, 0.6675)
7	5	6	(0.1153, 0.5947)
8	5	6	(0.7067, 0.3616)
9	5	6	(0.6293, 0.5094)

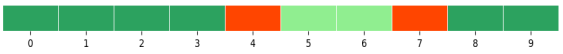

Figure 6: Top : Policy table extracted from Deep Q-Network in the Toy scenario. Cached items are highlighted with light gray color. Bottom : Policy Heatmap of the extracted policy in the Toy scenario.

extracted from Policy Iteration, its fair to assume that the above implemented algorithm does indeed converge to a nearly optimal policy. As stated before, states 5 and 6 cannot recommend themselves. Notably, the recommended items consistently include 5 and 6, which are the cached items, while also giving priority to the relevance of the recommended items. These results affirm the DQN's ability to consistently suggest relevant items, specifically 5 and 6, aligning with the predefined threshold, thus showcasing the effectiveness of the algorithm in providing suitable recommendations.

4.3. Algorithm Reward Analysis for Various Parameters

In order to evaluate and compare the performance of Policy Iteration and Q-Learning algorithms in our recommendation system, we conducted a series of experiments under different parameter settings. The experiments aimed to investigate the impact of varying values of K (number of items), u_{min} (relevance threshold), a (probability of choosing a recommended item), and q (probability of ending the session) on the convergence and effectiveness of the algorithms. We designed multiple experiments, each focusing on specific combinations of parameter values. The goal was to explore a wide range of scenarios and user behaviors to gain insights into the algorithms' performance under different conditions. We decided to measure the average reward accumulated by each algorithm throughout the episodes. For that reason, average reward in a given episode is calculated as follows:

$$R_{ep} = \frac{1}{S} \sum_{i=1}^{S} R_i$$

where R_i are all the rewards of the current episode, S are the total steps executed in the current episode. For the

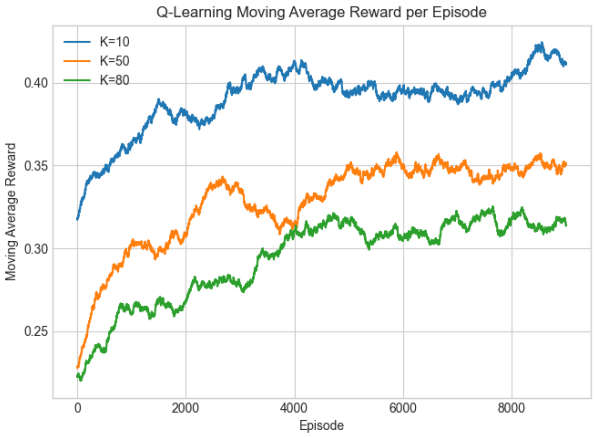

Figure 7: Q-learning moving average reward per episode for different number of items(K).

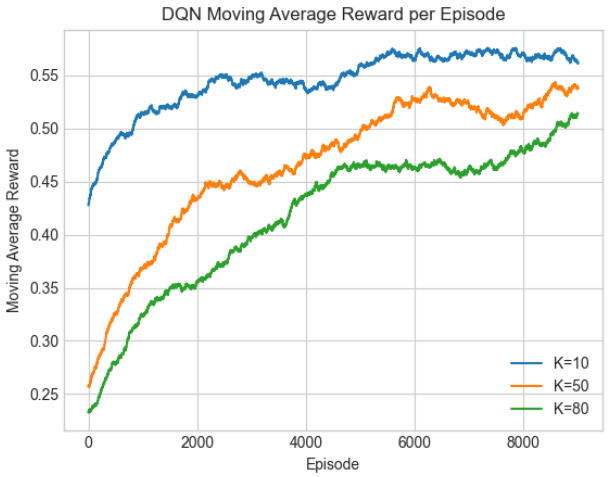

Figure 8: Deep Q-Network moving average reward per episode for different number of items(K).

variables that do not change, we used the default configuration parameters, as in toy example. For the evaluation of the policies, we used the average reward, i.e

$$R_{total} = \frac{1}{H}\sum_{i=1}^{H} R_{ep}(i)$$

where H is the total number of episodes, $R_{ep}(i)$ is the reward accumulated on a single episode i. The findings of the analysis are presented in Table 1. Below follows a summary of the experiments conducted.

4.3.1. Number of items (K)

For each experiment, we selected a specific value for K, representing the number of items in the content catalog. We varied this parameter to observe the scalability and efficiency of the algorithms for different dataset sizes. For this reason, in this particular experiment we ŕreezeṕarameter $N = 2$, in order to solely observe the impact of the total number of items K in the algorithms convergence. Observing the moving average reward per episode for Q-Learning and DQN in Fig. 7 and Fig. 8 respectively, we can comment that but algorithms can scale up to a certain amount of items. Both moving average rewards per episode seem to increase as number of items decreases. That is logical, since the algorithm facilitates to choose the rights recommendations when there are limited choices, and a constant number of recommendations (N). We will explore the scalability property of the algorithms with a different number of recommendations N in the next experiments.

4.3.2. Relevance Threshold (u_{min})

The relevance threshold (u_{min}) was another parameter of interest, which determined the level of relevance required for two items to be considered related. We experimented with different values of u_{min} to evaluate its impact on the recommendation quality. The impact of the relevance threshold in items relevance can be easily seen in Fig. 9. As the threshold increases, the relevant items decrease.

By modifying u_{min} we can model different profile of users about their relevance preferences. For the shake of the experiment, we can assume that $u_{min} = 0.1$ describes a **liberal user**, who may consider a broader range of items as relevant, showing a willingness to explore and engage with different suggestions. By setting the relevance threshold $u_{min} = 0.5$, we can model a **balanced user** who seeks a balance between relevant recommendations and a variety of options and values both precise relevance and a diversity of suggestions. Finally, $u_{min} = 0.9$, describes a **cautious user**. A cautious user is only interested in highly pertinent recommendations, ignoring or dismissing those that don't strongly align with their preferences or needs. In this experiment, we explore the effect of different type of users on each recommendation algorithm.

As we evaluate the performance of Q-Learning and Deep Q-Network for varying $u_{\min}$ values, a distinct trend becomes evident. Both algorithms demonstrate improved performance as the relevance threshold $u_{\min}$ decreases. For smaller values of $u_{\min}$,i.e for a liberal user, indicating a more lenient criterion for relevance, the algorithms can identify and recommend a wider range of items, consequently achieving higher cumulative rewards. This behavior aligns with the intuitive notion that a more permissive definition of relevance allows for a larger pool of potentially rewarding recommendations. However, as we increase $u_{\min}$,i.e getting closer to a cautious user, representing a stricter relevance criterion, both Q-Learning and DQN face challenges in identifying sufficiently relevant items. This leads to a decline in performance, with lower cumulative rewards, as the algorithms struggle to meet the heightened criteria for recommending items. Thus, the performance of the algorithms is inversely related to

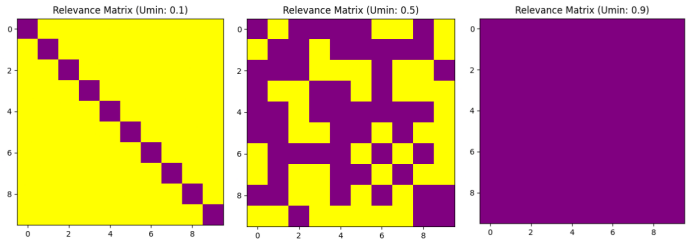

Figure 9: Heatmap of the relevance matrix ($K = 10$). Relevant items are described with yellow color. Irrelevant items are described with purple color.

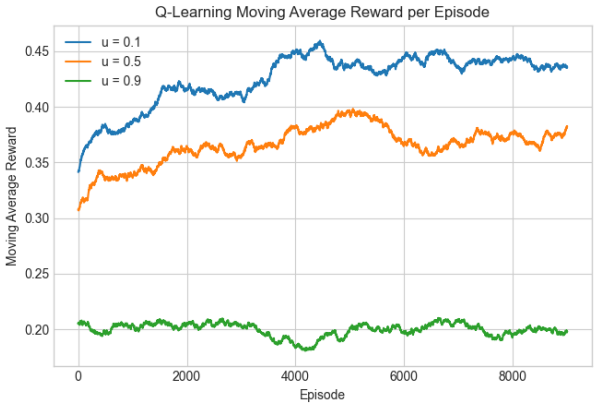

Figure 10: Q-learning moving average reward per episode for different relevance thresholds(u_{min}).

the value of $u_{\min}$, showcasing the critical role of relevance threshold in recommendation system performance.

4.3.3. User Probability to quit (q)

In our recommender system, the probability of user quitting (q) after viewing an item or receiving a recommendation plays a significant role in determining user engagement. A user with a low q value ($q = 0.1$) is more patient and less likely to quit the system quickly. He is willing to explore recommendations and give the system more chances to provide relevant items. The reward in this scenario tends to increase because the user engages more with the recommendations, giving the algorithm opportunities to learn from the user and suggest relevant items, ultimately resulting in a higher cumulative reward. A user with a moderate q value ($q = 0.5$) has a more balanced approach. They might explore the recommendations but may also quit after a few interactions. This could be due to varying levels of interest or limited time. The reward might be moderate since they interact with the system, but the chances of quitting are also higher compared to the case with $q = 0.1$. A user with a high q value ($q = 0.9$) is more likely to quit the system quickly. This indicates a user who has less patience or interest in exploring recommendations. They might quit after viewing just a few items, limiting the opportunities for the algorithm to learn suggest relevant items. Consequently, the reward tends to

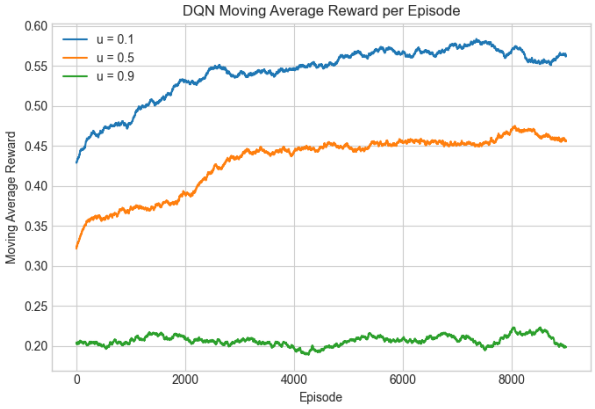

Figure 11: Deep Q-Network moving average reward per episode for different relevance thresholds(u_{min}).

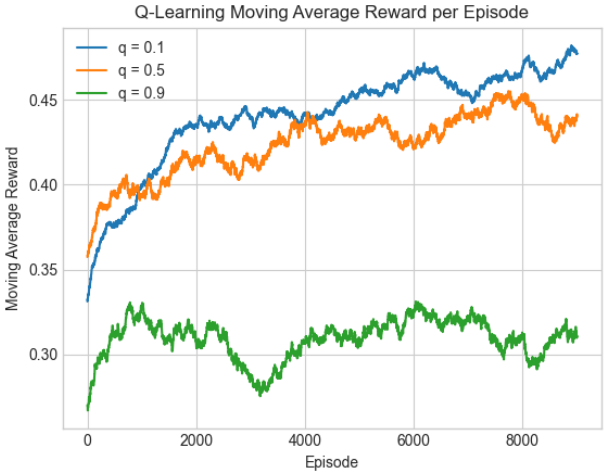

Figure 12: Q-learning moving average reward per episode for different probabilities of user to quit(q).

be lower in this scenario due to the limited interaction and reduced chance for the algorithm to make effective recommendations.

Observing the moving average reward plots in Fig. 10 and Fig. 11 we can identify similar effects of the probability of quitting on both recommendation algorithms. As the probability of quitting (q) decreases, the user tends to be more patient and engages more with the system, exploring recommendations and providing the algorithm with more opportunities to suggest relevant items. This leads to an increase in the cumulative reward. On the other hand, as q increases, the user is more likely to quit early, resulting in fewer interactions and a lower cumulative reward. This trend highlights the importance of user engagement in achieving higher rewards in recommender systems.

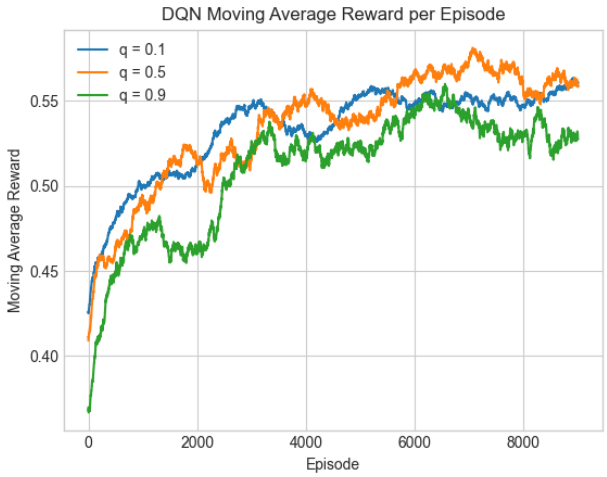

Figure 13: Deep Q-Network moving average reward per episode for different probabilities of user to quit(q).

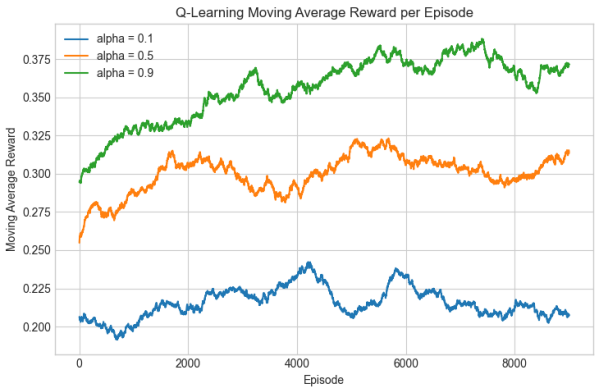

Figure 14: Q-learning moving average reward per episode for different probabilities of user to select a recommended item (*alpha*).

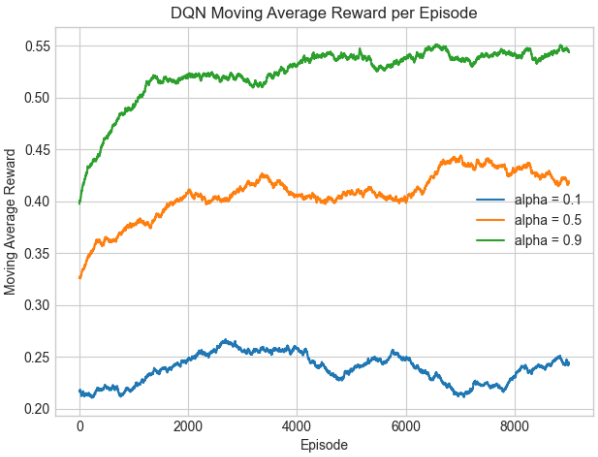

Figure 15: Deep Q-Network moving average reward per episode for different probabilities of user to select a recommended item (*alpha*).

4.3.4. User Probability to select a recommended item (alpha)

In the context of a recommendation system, the probability α represents the likelihood that a user will choose to watch a recommended item, given that all the recommendations provided are relevant. Let's discuss the characteristics of users for different α probabilities and explain why rewards tend to increase as α increases, as we can observe from Fig. 14 and Fig. 15.

A user with a low α value (e.g. α=0.1) is quite selective when it comes to choosing from the recommendations. Even if the recommendations are relevant, they are less likely to watch them, possibly indicating a cautious or discerning user. In this case, the reward tends to be lower as the user is less likely to engage with the recommendations, resulting in a reduced cumulative reward. A user with a moderate α value (e.g. α=0.5) strikes a balance in choosing recommended items. They are moderately likely to watch a recommended item if it's relevant. This could signify a user who evaluates recommendations fairly and is willing to engage with the system. The reward in this scenario is moderate, reflecting a balance between engagement and selectiveness. A user with a high α value (e.g. α=0.9) is more inclined to choose and watch recommended items when they are relevant. This implies a user who highly values the recommendations and is more likely to engage with the system. The reward tends to be higher in this case because the user engages more, resulting in increased interactions and a higher cumulative reward.

In summary, as the probability to choose a recommended item (α) increases, the user tends to be more receptive and engaged with the recommendations. Both algorithms again display similar effects, but DQN seems to surpass Q-Learning in term of average reward as shown in Table 1. A higher α indicates a user who appreciates and values the system's suggestions, leading to increased interactions and, consequently, a higher cumulative reward. Conversely, as α decreases, the user becomes more selective and less engaged, resulting in a lower reward.

4.4. Experiment: Convergence to Policy for "large" setups, DQN and Q-Learning .

In this experiment, we explore the convergence behavior of Q-Learning and Deep Q-Networks (DQN), in the context of larger recommendation environments. Testing the performance of the algorithms in larger setups involves evaluating their efficacy in scenarios with a greater number of recommendations (N). This scalability assessment aims to ascertain how well these algorithms adapt and perform as the system handles a larger set of potential recommendations. In such expansive environments, the number of possible actions grows considerably, requiring the algorithms to navigate a complex action space more effectively. The findings of this experiment are presented in Table 2.

Q-Learning encounters significant challenges when confronted with large action spaces, particularly when applied

	K			u_{min}			q			a		
Algorithm	10	50	80	0.1	0.5	0.9	0.1	0.5	0.9	0.1	0.5	0.9
Q-Learning	0.38	0.32	0.29	0.41	0.36	0.19	0.43	0.42	0.3	0.21	0.29	0.35
DQN	0.53	0.45	0.41	0.53	0.42	0.2	0.52	0.52	0.5	0.23	0.4	0.51

Table 1: Average Reward analysis of **Q-learning** and **Deep Q-Network** for different parameters values.

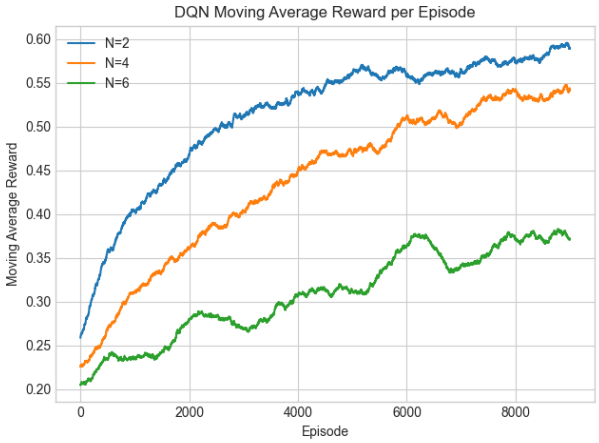

Figure 16: Deep Q-Network moving average reward per episode for different $K = 50$ and different number of recommendations (N).

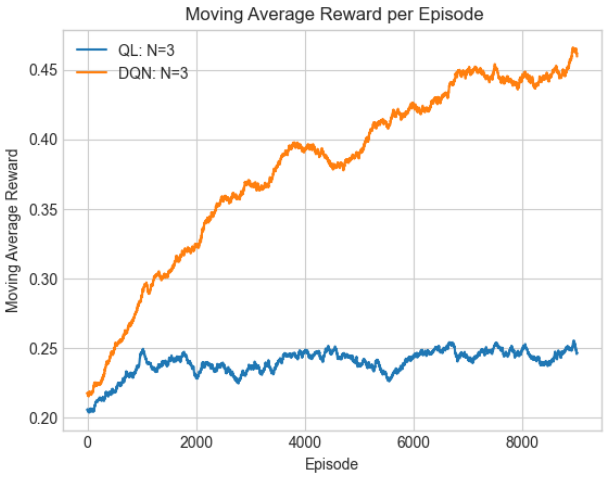

Figure 17: Comparison of Q-Learning and Deep Q-Network moving average reward per episode for $\mathbf{K = 100}$ and $\mathbf{N = 3}$.

to recommendation systems where the number of recommendations (N) is substantial. The fundamental issue lies in the exponential growth of the action space with the increase in N. In Q-Learning, the action space is determined by N^K , where K represents the number of items a user can choose from. As N rises, the action space grows exponentially, making it increasingly difficult to effectively explore and evaluate each possible action. This exponential growth hampers the learning process as the agent struggles to thoroughly assess a ballooning set of potential actions. Consequently, Q-Learning's efficiency and effectiveness diminish significantly as the action space becomes prohibitively large. In contrast, our DQN approach addresses this obstacle adeptly, ensuring that the action space increases linearly with N, allowing for more efficient navigation and assessment of potential recommendations. In Fig. 16, we show that our algorithm is capable to scale for a bigger number of recommendations. Its logical to assume that the lesser the number of the recommendations(N), the better the performance of the algorithm, since our system will have fewer items choices. By structuring our DQN to have an action space defined by the number of items to choose (K), we mitigate the exponential growth of the action space, enabling more manageable and effective learning in expansive recommendation systems. We could only compare DQN and Q-Learning in the $K = 100, N = 3$ scenario in Fig. 17, since Q-Learning imposing memory problems for larger action spaces.

- $K = 100, N = 3$
- $K = 500, N = 5$
- $K = 1000, N = 6$

Method	Setup	Reward	Time (s)
QL	(K=100, N=3)	0.23	229
	(K=500, N=5)	-	-
	(K=1000, N=6)	-	-
DQN	(K=100, N=3)	0.37	148
	(K=500, N=5)	0.2	200
	(K=1000, N=6)	0.19	296

Table 2: Final Cost and Execution Time for QL and DQN in different "large" setups

5. Conclusion

In conclusion,We conducted a comprehensive evaluation of Q-Learning (QL) and Deep Q-Network (DQN) across multiple scenarios. In smaller environments, both QL and DQN demonstrated proficient performance. However, as the environments grew in complexity and size, DQN emerged as the superior choice. It exhibited not only

swifter but also more accurate results in larger environments, underlining its efficiency in handling linearly expanding action spaces. On the other hand, QL encountered significant challenges, particularly in memory capability, hindering its successful execution in larger environments. In summary, our experiments consistently demonstrated that DQN outperforms Q-Learning across the board. This success underscores the superiority of neural network-based approaches, like DQN, when it comes to handling larger action spaces.