SlateQ for Slate-Based E-Commerce Recommender Systems

Muhammad Rasyid Gatra Wijaya

MSc in Computer Science
The University of Bath
2024/25

SlateQ for Slate-Based E-Commerce Recommender Systems

Submitted by: Muhammad Rasyid Gatra Wijaya

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Master of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

This dissertation explores reinforcement learning and standard recommender system baselines for slate recommendation in a simulated e-commerce environment. Traditional recommenders are designed for single-item suggestions, which makes it harder to capture long-term user engagement and the combinatorial nature of slates. The project uses Google's `RecSim NG` framework to benchmark SlateQ and its variants against common baseline agents within an environment that simulates multiple users, a fixed item catalogue, and simple choice behaviour.

Four SlateQ variants (vanilla, duelling, NoisyNet, and a combined Dueling+NoisyNet) were compared with DQN, contextual bandit, greedy, and random baselines. Performance was measured using cumulative reward, training stability, and ranking quality metrics such as NDCG@5 and Slate MRR. SlateQ vanilla achieved the highest overall reward (≈ 2533 in the final 100 episodes), which shows its reliability for cumulative engagement. NoisyNet gave the strongest ranking quality (MRR ≈ 0.019) but sacrificed some reward in return, while the duelling version performed worse than expected. The combined variant recovered part of the lost reward and maintained good ranking quality but showed unstable training. DQN delivered competitive reward but weaker rankings, which demonstrates the limits of standard deep RL. Surprisingly, simple baselines like greedy and contextual bandits sometimes matched or exceeded SlateQ in reward but produced weaker ranking signals. Overall, SlateQ achieves a strong balance between reward and ranking, while strong baselines remain valuable points of comparison.

Contents

List of Figures

List of Tables

Acknowledgements

Above all, I thank Allah (SWT) for giving me the strength and patience to complete this work. I am also grateful for the opportunities and guidance that have helped me reach this point.

I would like to thank my supervisor, Andreas Theophilou, for his advice and constant support throughout this project. I am very thankful to my parents and family for their prayers, encouragement, and endless support. I also want to thank my friends for keeping me motivated and helping me stay balanced along the way.

Lastly, I would like to thank everyone who supported me in any way during this dissertation, whether directly or indirectly. Your help has been truly appreciated and has made this journey possible.

Chapter 1

Introduction

1.1 Motivation

Traditional recommendation models often focus on suggesting a single item at a time. One common strategy is short-term optimisation, such as fixed greedy policies based on immediate feedback. While effective in the short term, these approaches can fail to capture user preferences that develop over time, which may limit long-term engagement (Zhao et al., 2018). In contrast, most modern systems such as e-commerce platforms, social media, or streaming services recommend a group of items at once, commonly referred to as a slate. This shift introduces a more complex action space, as the number of possible item combinations grows exponentially.

The limitations of short-term optimisation and the complexity of slate-based recommendation challenges have motivated recent research into reinforcement learning approaches that aim to optimise long-term user outcomes (Chen et al., 2023) and address the combinatorial complexity of slate-based recommender systems (Ie et al., 2019b). Because of this, Reinforcement Learning (RL) is increasingly seen as a good fit for slate-based recommendation problems. Unlike contextual bandits, which treat each interaction independently, full RL agents can learn sequential decision policies that account for user history and adapt based on long-term outcomes (Li et al., 2010a). This enables systems to move beyond short-term optimisation and better capture dynamic user preferences over time (Dulac-Arnold, Mankowitz and Hester, 2019). However, applying RL to recommendation tasks introduces significant scalability challenges. Deep Q-networks (DQNs), for example, have demonstrated strong performance in high-dimensional control tasks such as Atari gameplay (Mnih et al., 2015a), but they are not suited for slate recommendation problems, where the number of possible item combinations is extraordinarily large (Ie et al., 2019b).

The SlateQ algorithm (Ie et al., 2019b) addresses this issue by decomposing the slate-level Q-value into per-item components, assuming user choice is conditionally independent across items. This enables efficient Q-learning in environments with large action spaces. Although SlateQ was initially evaluated in RecSim (Ie et al., 2019a), there has been little independent replication or comparison with standard baselines such as DQN or contextual bandits. Furthermore, few studies have explored improvements to SlateQ, such as diversity-aware rewards or enhanced user models. This project explores these gaps by extending SlateQ, testing new techniques, and benchmarking it against other RL methods focused on long-term engagement, specifically in e-commerce settings.

1.2 Aims and Objectives

This research aims to:

- Assess how well SlateQ performs in slate-based recommendation environments, with a focus on long-term user engagement and adaptive behaviour.
- Compare reinforcement learning methods, including SlateQ, with simpler baselines to see their strengths in modelling sequential user preferences.

The objectives are to:

- Implement and train a SlateQ agent in RecSim NG within a simulated e-commerce recommendation setup.
- Benchmark SlateQ's performance against its duelling and NoisyNet variants, as well as DQN, contextual bandits, greedy, and random agents.
- Analyse where each algorithm performs well or struggles in different recommendation settings.
- Explore tuning options and possible extensions, such as diversity-aware rewards, to improve SlateQ.
- Consider the practicality of using SlateQ in real-world recommendation systems.

Chapter 2

Literature and Technology Survey

2.1 Traditional Recommender Systems

Recommender systems aim to anticipate user preferences and suggest items that align with those interests. The earliest approaches are commonly divided into two categories: content-based methods and collaborative filtering. Content-based systems recommend items that share attributes with those a user has previously engaged with, typically by comparing item features against a user profile. Collaborative filtering (CF), in contrast, exploits patterns across user and item interactions. In practice, this involves recommending items that similar users have consumed, or suggesting items related to those a user has already selected. Early memory-based CF methods, such as user-based and item-based neighbourhood models, were among the first to achieve widespread use (Adomavicius and Tuzhilin, 2005). Subsequently, model-based approaches such as matrix factorisation became influential, as they embed users and items into a shared latent space. The Netflix Prize is a well-known example where matrix factorisation demonstrated considerable success in capturing subtle user and item relationships (Koren, Bell and Volinsky, 2009). Approaches of this kind typically generate a ranked list of individual items. Each item is assigned a relevance score, for example a rating prediction or a click-through probability, and the list is then ordered from most to least relevant.

As research progressed, more sophisticated machine learning and deep learning techniques were introduced. Factorisation machines and deep neural networks have been applied to capture higher-order interactions between features, while recurrent neural networks have been used for sequential or session-based recommendation, where the order of user actions is significant. Zhang et al. (2019) provide a comprehensive survey of recent developments and emphasise the growing impact of deep learning in recommender systems. Despite these advances, traditional recommenders generally rank individual items independently. Each recommendation is made in isolation, with the system selecting the top-N items that appear most relevant at a given time. While effective for short-term suggestions, this approach overlooks how items may complement one another within a slate, and does not account for how current recommendations can shape future user preferences. These limitations have motivated a shift towards frameworks that consider groups of items together and model recommendation as a sequential decision-making process.

2.2 The Shift to Slate-Based Recommendation

In practice, most modern platforms do not recommend items one by one but present a set of options at the same time. E-commerce websites, social media feeds, and streaming services typically show users a slate of items, such as a page of product suggestions or a grid of video thumbnails. This design makes sense: offering multiple choices increases the chance that the user finds something they like. However, it also introduces new complexity. A slate is not just a collection of individual recommendations. The items within a slate can influence one another, shaping how the user perceives and selects from them. For instance, showing too many similar items might make the page feel repetitive, while a more varied slate could capture different aspects of the user's interests. Choosing the best slate isn't just about picking the top items, as you also need to think about diversity, balance, and order.

Moving to slate-based recommendations also makes the action space dramatically larger. Instead of choosing a single item from a catalogue of N items, the system must choose a combination of k items. The number of possible slates grows rapidly with N and k. For example, if $N = 20$ and $k = 5$, the number of ordered slates is:

$$P(20, 5) = \frac{20!}{(20-5)!} = 20 \times 19 \times 18 \times 17 \times 16 = 1{,}860{,}480 \approx 1.86 \times 10^6$$

Even this relatively small example results in nearly two million possible slates. With larger catalogues, the number of combinations quickly becomes unmanageable and makes exhaustive search impossible.

To cope with this, researchers and industry teams have developed algorithms that work directly at the slate level. One example is page-wise recommendation, where the system selects a group of items together rather than one at a time (Zhao et al., 2018). By treating the slate as a single decision, these methods account for how items interact within the set and aim for broader goals, such as improving user satisfaction with the whole page or ensuring a balanced mix of content. This shift has made the problem more challenging but also opened the door to stronger decision-making approaches, including reinforcement learning.

2.3 Challenges in Slate Recommendation

Slate-based recommendation introduces challenges not present in traditional systems:

- **Combinatorial Action Space:** The number of possible slates grows rapidly with catalogue size and slate length. Exhaustive search is infeasible, and simple greedy strategies risk redundancy. Effective methods must approximate the best slate without brute force (Ie et al., 2019b).
- **Long-Term User Dynamics:** Recommendations occur across sessions and visits, so focusing only on immediate clicks can harm long-term engagement. Repeating similar items may work once but soon becomes stale (Zhao et al., 2018). Capturing how today's slate shapes tomorrow's preferences requires sequential models (Dulac-Arnold, Mankowitz and Hester, 2019).
- **Efficiency and Scalability:** Real-world systems serve millions of users and items, so algorithms must be both fast and scalable. Training on large datasets or simulations

is resource-heavy, and practical solutions rely on pruning or structured approximations (Dulac-Arnold, Mankowitz and Hester, 2019; Ie et al., 2019b).

These difficulties have pushed researchers towards reinforcement learning (RL), which is well-suited for sequential decision-making in large action spaces. Still, applying RL to slates is not straightforward, as it must balance combinatorial complexity with realistic models of user behaviour. Before turning to specialised methods such as SlateQ, we first review how RL has been applied to recommender systems in general.

2.4 Reinforcement Learning in Recommender Systems

Reinforcement learning (RL) treats recommendation as a process that happens step by step. At each interaction, the system suggests one or more items based on what it currently knows about the user, which could include their profile, browsing history, or current situation. After making a recommendation, the system receives feedback from the user, such as a click or a purchase, which it uses to learn and update its understanding of the user's interests. This kind of setup aligns well with the Markov Decision Process (MDP), where the aim is to find a strategy that earns the most total reward over time, instead of just focusing on each recommendation separately.

Early uses of reinforcement learning in recommendation were held back by limited computing power. Shani, Heckerman and Brafman (2005) introduced MDP-based recommenders that track how users move between different states and aim to improve future rewards. However, these early systems usually worked only in simple settings, with a small number of choices and short time spans. A big improvement came with contextual bandits, basic versions of RL that pick one item at a time, based only on the current situation, without planning for the future. Li et al. (2010a) showed that this approach could work at scale by deploying a contextual bandit on Yahoo!'s Front Page using the LinUCB algorithm, which finds a balance between recommending familiar content and exploring new options. While effective, contextual bandits are not ideal for slate recommendation, since they usually recommend just one item at a time and don't account for how today's choice might affect what the user wants tomorrow.

More recently, deep reinforcement learning (DRL) has been used to deal with more complex user information and large sets of possible recommendations. DRL uses neural networks to help the system learn which actions (like which items to show) are best in different situations. Methods like Deep Q-Networks (DQN) and policy gradient approaches have shown good results in tasks where the system needs to make recommendations over many steps. Dulac-Arnold, Mankowitz and Hester (2019) point out that there are still challenges, such as not knowing everything about the user or learning from past data, but there have been real successes. For example, Ie et al. (2019b) mention how a DQN-like method was used in YouTube's recommendation system, showing that RL can work even at a very large scale.

2.5 The SlateQ Algorithm

SlateQ, introduced by Ie et al. (2019b), is an RL algorithm designed to handle the combinatorial nature of slate-based recommendation in a tractable way. In typical recommender environments, users are presented with a *slate*, a set of multiple items, rather than a single recommendation.

SlateQ addresses the exponentially large action space challenge by introducing a decomposition

technique that breaks down the slate-level Q-function $Q(s, A)$, the expected cumulative reward from presenting slate A in state s, into a weighted sum of item-level Q-values $Q(s, i)$. This relies on two main assumptions: users select only one item from the slate (Single Choice), and both reward and state transitions depend solely on the consumed item rather than the entire slate (RTDS: Reward and Transition Dependence on Selection).

Formally, under these assumptions and using a choice model such as the Multinomial Logit, the value of a slate A can be approximated as:

$$Q(s, A) = \sum_{i \in A} P(i \mid s, A) \cdot Q(s, i)$$

where $P(i \mid s, A)$ is the probability of a user selecting item i from slate A in state s. This transforms the combinatorial slate optimisation into a linear programming problem, or even simpler greedy heuristics.

For training, the item-level Q-values are updated using a decomposed temporal difference rule:

$$Q(s, i) \leftarrow \alpha \left(r + \gamma \sum_{j \in A'} P(j \mid s', A') Q(s', j) \right) + (1 - \alpha) Q(s, i)$$

where A' is the next slate and s' the subsequent state. At serving time, the agent constructs a slate by selecting the top-k items with the highest $v(s, i) \cdot Q(s, i)$ scores, where $v(s, i)$ is an unnormalised proxy of the user's interest (for example, from a click-through-rate model).

This decomposition enables generalisation across slates, improves sample efficiency, and is easily integrated into systems like YouTube, where SlateQ was deployed in a live experiment. The algorithm significantly improved user engagement over baseline myopic recommenders, which demonstrates its scalability.

2.6 Baseline Reinforcement Learning Algorithms

To contextualise SlateQ's performance, this project also compares it against two commonly used reinforcement learning baselines, in addition to simple Greedy and Random policies:

- **DQN (Deep Q-Network):** A value-based method that uses neural networks to estimate action values (Mnih et al., 2015b). In recommendation settings, DQN recommends one item per step and learns which choices maximise long-term reward.
- **Contextual Bandits (LinUCB):** A simplified approach that selects items based on the current user context without modelling long-term effects (Li et al., 2010b). LinUCB balances exploration and exploitation and is widely used for short-term metrics such as click-through rate.

2.7 Evaluation Framework: RecSim and RecSim NG

Evaluating RL algorithms for recommender systems in real-world settings is both challenging and risky. Live experiments can be slow, expensive, and potentially harmful to user experience if an underperforming policy is deployed. To avoid these issues, researchers rely on simulation

environments. One such tool is RecSim, an open-source platform developed by Google for modelling recommender environments (Ie et al., 2019a).

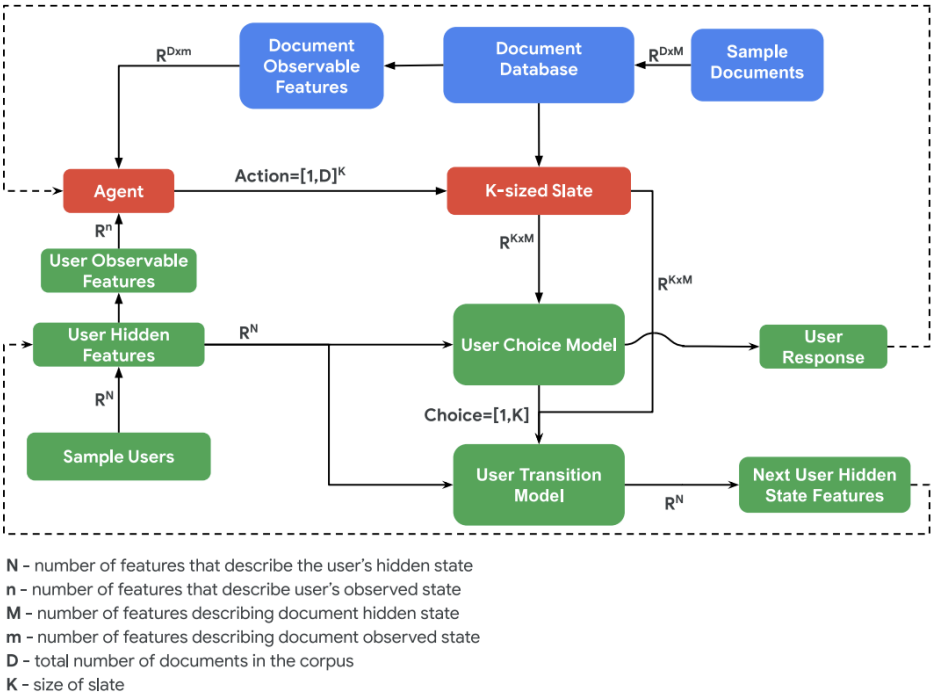

Figure 2.1: RecSim framework for simulating user and item interactions (Ie et al., 2019a).

RecSim provides a configurable RL environment where a recommender agent interacts with a pool of documents (items) and user profiles. As shown in Figure 2.1, the setup typically includes a user model, a document model, and a user choice model. The agent proposes a slate of items, the user model decides which item (if any) is consumed, and the user's state is updated accordingly. User models can be defined with latent features such as interests and satisfaction, as well as observable traits like demographics or time budget. Document models capture item attributes including quality, topic, or popularity. This flexibility makes RecSim useful for testing a wide range of research ideas in a safe and reproducible way. Its main strength is treating recommendation as a sequential decision problem, which makes it well suited to studying long-term engagement and evolving preferences.

While RecSim remains a solid platform, it has limitations when experiments require complex customisation. To address this, Google released RecSim NG (Next Generation), a graph-based simulator for recommender research (Ma et al., 2021). RecSim NG builds on the same core idea as RecSim but offers a more modular and expressive design. Environments are represented as probabilistic graphical models composed of `Variable` nodes and dependencies, which makes it easier to define user dynamics, item attributes, and recommender logic in a clean, extensible way. This structure also integrates naturally with TensorFlow and supports batched execution as well as compatibility with RL frameworks such as TF-Agents.

Another advantage of RecSim NG is its explicit support for uncertainty modelling. By representing user behaviour and item responses probabilistically, it allows researchers to test algorithms under more realistic conditions, where outcomes are not deterministic. The framework is designed with reproducibility in mind and makes it easier to compare approaches

across experiments. Although RecSim NG is not a full-scale production simulator, it provides a principled middle ground between simple testbeds and costly online trials.

In this project, RecSim NG is used as the evaluation framework. Its flexibility and clean API made it straightforward to model a simplified e-commerce setting, define user choice behaviour, and run experiments with multiple agents. The ability to capture sequential interaction and long-term outcomes, while keeping experiments reproducible and safe, makes RecSim NG well suited to benchmarking algorithms like SlateQ and its extensions.

2.8 Research Gaps

Reinforcement learning for slate-based recommendation is promising, but several gaps remain that this project addresses:

- **Benchmarking:** Prior work on SlateQ Mladenov et al. (2021) mainly evaluates the vanilla algorithm against simple baselines in limited setups. This project benchmarks several RL algorithms, including SlateQ and its variants, DQN, LinUCB, Greedy, and Random, within the same RecSim environment using a consistent setup for a fair comparison.
- **Diversity:** SlateQ does not explicitly optimise for diverse slates. We examine whether long-term planning naturally encourages variety or tends to focus on similar items.
- **Reproducibility:** Using RecSim NG and reporting standardised results improves reproducibility and provides a baseline for future slate-based studies.
- **Future Improvements:** The project considers potential extensions, such as diversity-aware rewards and richer user models, as directions for future work.

Chapter 3

Methodology

This chapter explains how the experiments in this project were designed and carried out. It covers the setup of the RecSim NG environment, the agents tested, the evaluation metrics used, and the experimental process followed to keep the results fair and reproducible.

3.1 RecSim NG Environment Setup

RecSim NG was chosen over the original RecSim because it offers greater flexibility when customising the environment (Ma et al., 2021). It provides a cleaner, graph-based API that makes it straightforward to define dependencies between components and incorporate custom modules without relying on complex workarounds. The `Network` setup allows the item, recommender, and user states to be treated as separate components, each with their own update rules, which simplifies both debugging and future modifications. This design is well suited for benchmarking since it enables different user choice models, additional item features, or changes to slate size to be introduced without rebuilding the entire environment. It also integrates smoothly with TensorFlow's functional style and supports a direct connection with TF-Agents. As a result, the full training loop covers action selection, slate construction, user choice, and reward calculation, and operates as a consistent and efficient batched process. The structure helps maintain consistency across experiments and reduces the risk of introducing unintentional biases.

The environment is composed of three main state models: item, recommender, and user, along with network assembly and runtime integration, as illustrated in Figure 3.1. These components work together to simulate the recommendation process, from generating candidate items to capturing user interactions and recording rewards.

3.1.1 Item model

The item state is static and holds a fixed set of feature vectors, one for each item in the catalogue. These vectors are generated at the start of the simulation from a normal or uniform distribution and represent the item's position in a latent topic space. Since the features never change, the item state only needs to be initialised once. A custom field specification defines the shape and type of the feature tensor to ensure consistency with other components.

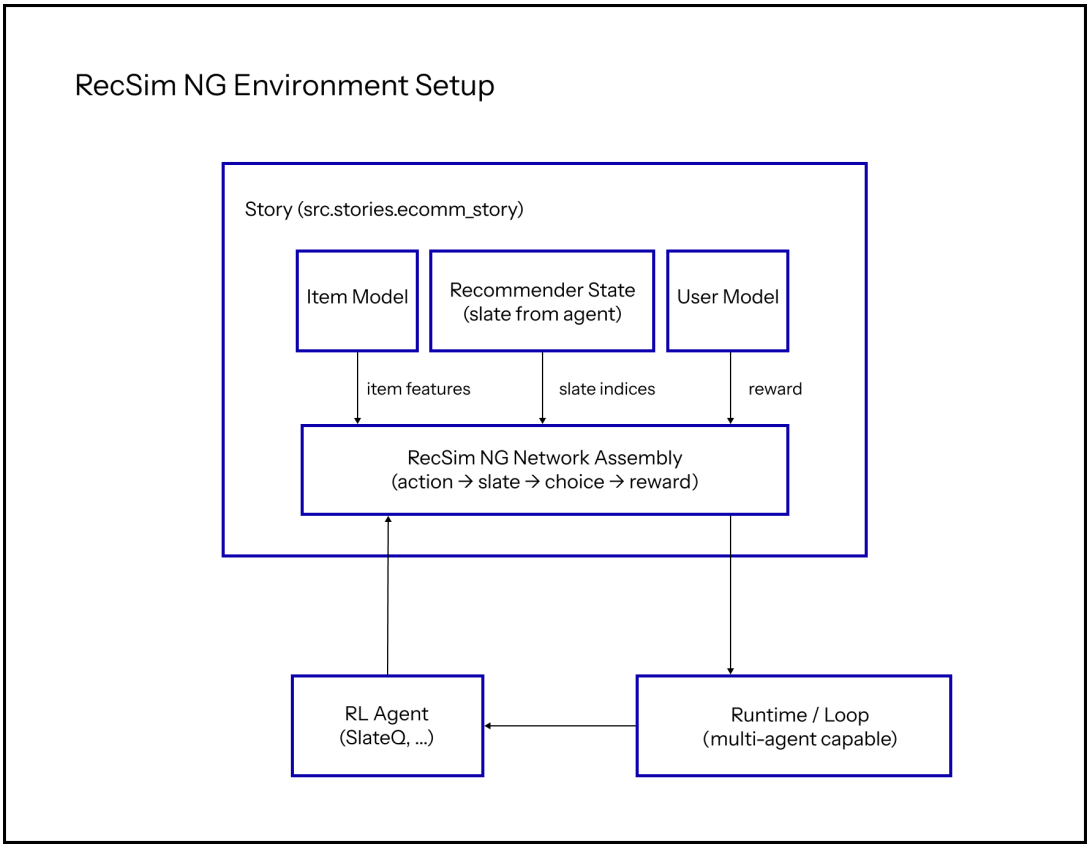

Figure 3.1: RecSim NG environment setup architecture

3.1.2 Recommender state

The recommender state stores both static recommender features and the slate of item indices chosen for each user. The features are initialised from a uniform distribution and remain constant throughout the run, while the slate is updated each step based on the agent's actions. The recommender defines an explicit action specification so that any agent interacting with the environment must produce slates with the correct dimensions, helping to avoid shape mismatches or invalid indices.

3.1.3 User model

Each user is represented by an interest vector, the index of the item they choose from the slate, and a reward value. Interests are initialised from a normal distribution and determine how well each item matches the user's preferences. At each step, the user model receives the current slate and item features, calculates an affinity score for each item in the slate, and selects exactly one item to consume. The reward is taken as the affinity score of the chosen item, simulating a simple but stochastic choice process.

3.1.4 Network assembly

These components are combined in a RecSim NG `Network` definition, which specifies the variables, their initial values, and how they are updated each step. The variables are linked through explicit dependencies so that the simulation follows a consistent sequence: the agent's action updates the slate, the user model generates a response, and the user state is updated

accordingly. Item and recommender features remain fixed, while the user's choice and reward change over time. This modular design makes it straightforward to swap in different agents or adjust the environment without rewriting the entire system.

3.1.5 Runtime integration

A custom runtime class wraps the network and adapts its inputs and outputs to match the `TimeStep` format expected by TF-Agents. This allows reinforcement learning agents such as the SlateQ implementation in this project to interact with the environment directly. The runtime also provides utility functions for resetting the simulation, stepping through interactions, and running extended rollouts for evaluation.

3.2 Agent Types

Several agent architectures were evaluated in the environment:

- **SlateQ (vanilla)**: the standard value decomposition approach. We estimate per-item Q-values with a deep Q-network (no duelling, no NoisyNet), then combine them to score a slate and take the top-K. Training uses a target network to stabilise updates and epsilon-greedy exploration with decay. Optionally weight by user-item affinity during ranking.
- **Random policy**: samples items uniformly at random to fill the slate. A sanity-check lower bound.
- **Greedy**: a deterministic baseline that sorts items by a simple score (usually user-item affinity) and picks the top-K. No learning or exploration.
- **Contextual Bandits**: short-horizon baseline that learns a linear scorer from the current user context only. Serves top-K with optional epsilon-greedy exploration.
- **DQN**: a vanilla per-item deep Q-network. Uses a target network, Huber loss, gradient clipping, reward scaling, and epsilon-greedy with exponential decay. For slate actions we take the top-K items by Q.
- **SlateQ (duelling)**: same setup but with a duelling head (value and advantage). Still uses a target network and epsilon-greedy decay. Ranks items into a top-K slate using Q and user-item affinity.
- **SlateQ (NoisyNet)**: SlateQ with NoisyNet layers for parameter-noise exploration. Keeps the target network. Exploration is driven by the noisy layers rather than just epsilon.
- **SlateQ (duelling + NoisyNet)**: combines a NoisyNet trunk with a duelling head to improve both exploration and credit assignment. Uses a target network and relies on parameter noise for exploration instead of epsilon.

Each agent interacted with the same environment setup and was trained under identical conditions, with hyperparameters tuned individually. This allowed for a fair comparison of their strengths and weaknesses in a slate-based recommendation setting.

3.3 Evaluation Metrics

At the end of each episode, a set of core metrics was recorded to evaluate agent performance from different perspectives:

- **NDCG@k** (Normalised Discounted Cumulative Gain): a widely used metric for assessing how closely the recommended slate matches the ideal ranking for a user. It considers both the relevance of each item and its position within the list and places higher weight on relevant items placed earlier. The score is normalised against the DCG of the ideal slate and always falls between 0 and 1 (Järvelin and Kekäläinen, 2002). In this setup, relevance is derived from simulated user clicks, and k is set to match the slate size.
- **Slate MRR** (Mean Reciprocal Rank): measures the position of the first relevant (clicked) item within the slate and averages the result across all users and episodes. For a single recommendation, it is calculated as $1/r$, where r represents the position of the clicked item. Higher values indicate that relevant items appear earlier in the slate (Voorhees and Tice, 1999).
- **Cumulative reward**: the total reward accumulated across all users in an episode. In RecSim NG, this acts as a proxy for overall engagement and reflects behaviours such as clicks or dwell time. It serves as the primary reinforcement learning objective.
- **Loss**: the training loss from the agent's learning process. It is used to check whether learning remains stable and whether convergence is achieved. For value-based approaches such as SlateQ, this corresponds to the temporal-difference (TD) loss averaged over the sampled mini-batches.

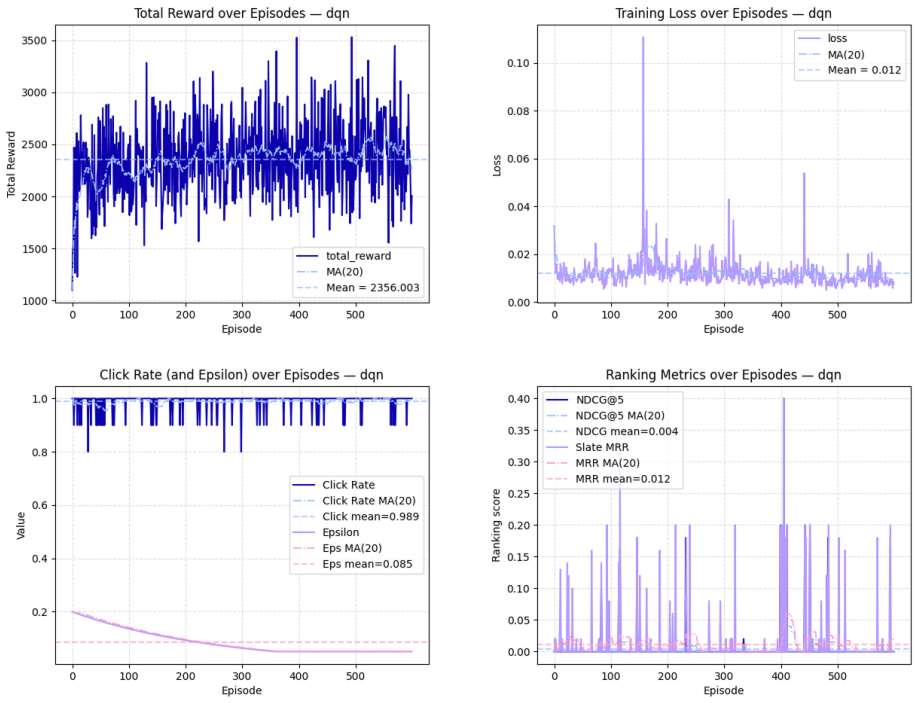

Figure 3.2: Illustrative example of the tracked metrics during training.

Including Figure 3.2 here makes it easier to recognise and interpret the plots that appear in

Chapter 5.

3.4 Experimental Procedure

To ensure the experiments were consistent and reproducible, the following setup was used:

- Fixed random seeds were applied for environment creation, agent initialisation, and training runs. This helped reduce variance between trials and made the results easier to compare.
- All agents were trained for the same number of episodes (600) with a fixed number of steps per episode. This length was chosen to allow agents enough time to converge while keeping training computationally feasible. Pilot runs indicated that most learning curves stabilised within this range, so extending beyond 600 episodes produced limited additional insights relative to the extra compute required.
- Hyperparameters (such as learning rate, discount factor, replay buffer size, and exploration schedule) were tuned manually for each agent through small pilot runs and adjustments. The aim was to keep training stable and ensure each agent had a fair chance to perform well, rather than to find the absolute best settings. Despite these differences, the overall training budget and evaluation conditions were kept identical across agents.
- Metrics were logged in both CSV and JSONL formats, making the results easy to analyse, plot, and reproduce. Plots were generated for total reward, training loss, and ranking metrics to give a balanced view of performance.

All experiments were carried out on CPU. GPU acceleration was not used due to time constraints and to keep the setup simple. This also reflects the practical constraint of limited compute resources.

Chapter 4

Implementation

This chapter describes the practical implementation of the simulation environment, agents, and training pipeline used in the experiments. It details the system architecture, environment design, agent integration, and logging mechanisms, and explains how the design choices from Chapter 3 were translated into a working system.

4.1 System Architecture

The project was structured as a modular Python package, with separate components for environment modelling, agent logic, runtime execution, and metric logging. Figure 4.1 provides a high-level view of the system's modules and their interactions.

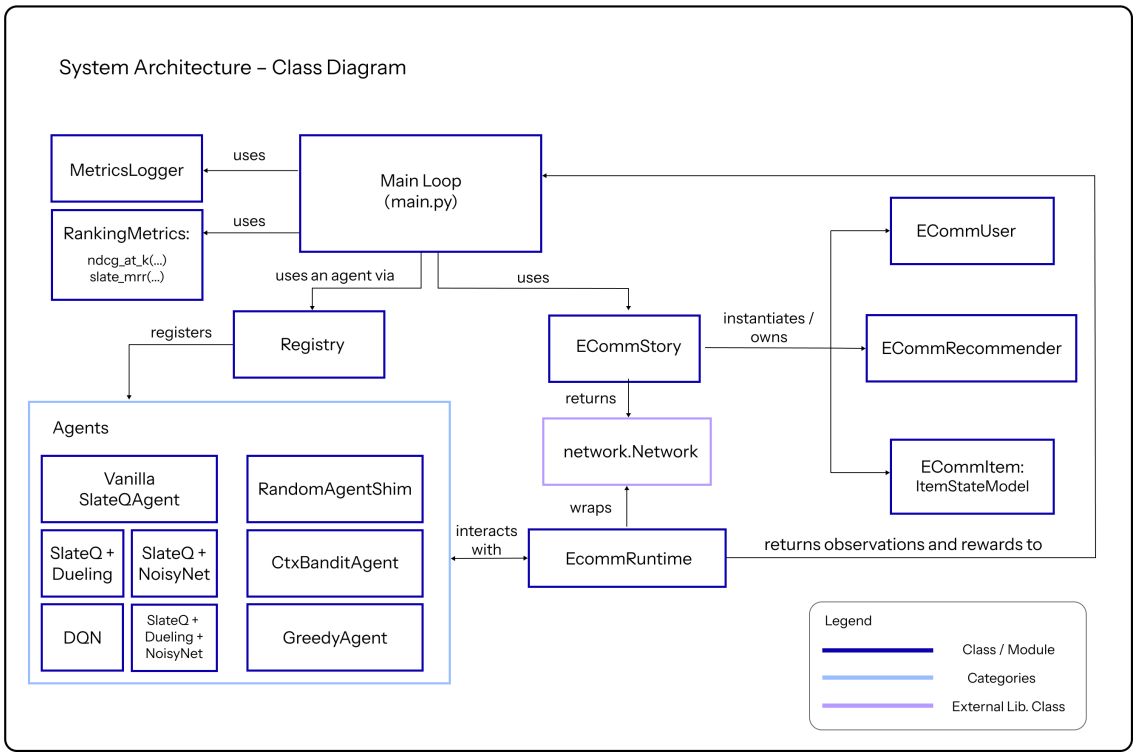

Figure 4.1: High-level system architecture for the simulation and training pipeline.

As shown in Figure 4.1, the `main.py` script acts as the primary control loop. It constructs the simulation by instantiating an agent via the `Registry`, sets up the environment through `ECommStory`, and passes it to the `ECommRuntime` for execution. `ECommStory` is responsible for composing the RecSim NG `Network` from environment entities such as `ECommUser`, `ECommRecommender`, and `ECommItem`. The `ECommRuntime` wraps the `Network` to facilitate stepping through time and exchanging actions, observations, and rewards with the agent. Performance metrics are computed via `RankingMetrics` and logged by `MetricsLogger` for later evaluation.

4.2 Environment Modelling

The environment was implemented using `RecSim NG`, a flexible framework for constructing simulation environments as directed acyclic graphs of `Variable` nodes. In this project, the environment is composed through the `ECommStory` function, which connects the user, recommender, and item state models into a single `Network` instance. Each subcomponent is implemented as a `StaticStateModel` to keep the simulation deterministic unless explicitly modified by agent actions or user responses.

4.2.1 ECommStory

The `ECommStory` function defines the full environment by instantiating the `ECommUser`, `ECommRecommender`, and `ECommItem` components, and linking them via `Variable` nodes. It specifies the `ValueSpec` for each variable (e.g., action, user state, response, recommender state, item state) and sets up initial values and transition functions. The `response` variable is produced by the user model in reaction to the recommender's slate and the current item state, while the user and recommender states are updated according to their respective `next_state` methods. Finally, `ECommStory` returns a `Network` object, which is later wrapped by the runtime for interaction with agents.

4.2.2 ECommUser

The `ECommUser` model simulates non-myopic users with long-horizon dynamics. Each user holds an interest vector over latent topics, initialised from a standard normal distribution and clipped for stability. On each slate, item affinities are computed and a click is sampled with a position-biased cascade that also allows a no-click outcome. The reward combines a clicked-item signal (sigmoid or normalised affinity), a diversity bonus over the slate, a penalty for repeated topical exposure, and any matured delayed conversions from a small queue. The model returns `choice`, `reward`, and a `continue_flag` for logging and control.

State transitions are explicitly non-myopic. Interests evolve through exposure-driven learning, fatigue, and forgetting, with optional small drift, followed by clipping and optional L2 renormalisation. A click pulls interests toward the chosen topic mix and applies mild saturation or novelty shaping. The model maintains recent exposure counts, an exposure streak used for a spaced-exposure conversion gate, a queue of pending conversions with stochastic delays, a novelty momentum term, and a decayed satisfaction logit. Continuation is determined by a logistic mix of diversity, novelty, mean affinity, momentum, and satisfaction, or forced to continue for training stability when enabled.

4.2.3 ECommRecommender

The `ECommRecommender` model defines the recommender's internal state, which includes a set of static feature vectors (one per user) and the current slate of recommended items. Its `specs` method defines the shape and type of these fields, and the `initial_state` method assigns random uniform values for the feature vectors with a zero-filled slate. While `next_state` simply returns the existing state in this static version, the `action_spec` defines the expected shape of the recommender's slate so that agents can interact with the environment consistently.

4.2.4 ECommItem

The `ECommItem` model represents the pool of available items in the simulation. Each item is associated with a static feature vector, sampled from a standard normal distribution across the defined number of topics. As a `StaticStateModel`, `ECommItem` does not change over time, and its `next_state` method simply returns the existing features. These item features are used by the `ECommUser` when calculating affinities and making choices from the slate.

4.3 Agent Implementations

The project implements a range of agents, from simple baselines to more advanced reinforcement learning models. All agents are instantiated via the `Registry` module, which maps string identifiers to constructor functions. This design makes it easy to switch between agents in the training loop by changing a single configuration parameter, without touching the core execution code.

4.3.1 Registry

The `Registry` follows a factory pattern and stores agent names along with their constructors through the `@register` decorator. This design keeps the system modular and makes it possible to add new agents with minimal changes.

4.3.2 SlateQ Agent Implementation

All SlateQ variants are implemented in `TF-Agents` with a shared training pipeline: Double Q-learning with target networks, Huber loss, gradient clipping, and Polyak updates. Each agent predicts per-item $Q(s, i)$ values and ranks items into a top-K slate by combining Q with user-item affinity scores.

Vanilla SlateQ

Implemented in `VanillaSlateQNetwork`, the base agent uses a three-layer MLP with hidden sizes $[512, 256, 128]$, ReLU activations, and layer normalisation. The Q-head outputs one value per item. Slates are constructed by ranking the learned $Q(s, i)$ values multiplied by cosine-style user-item affinity:

$$\text{score}(i) = Q(s, i) \cdot \frac{\mathbf{u} \cdot \mathbf{f}_i}{\|\mathbf{u}\|\|\mathbf{f}_i\|}$$

Exploration follows an ϵ-greedy policy with exponential decay, handled by `SlateQExplorationPolicy`. Targets are computed as a position-weighted expectation over the next greedy slate.

Duelling SlateQ

This variant modifies the Q-network to separate state value and item advantage:

$$Q(s,i) = V(s) + A(s,i) - \frac{1}{N}\sum_{j} A(s,j)$$

The implementation adds two parallel Noisy or Dense heads: one for $V(s)$ (scalar) and one for $A(s,i)$ (per item). This improves credit assignment within slates while keeping the same replay buffer and target computation.

NoisyNet SlateQ

The dense layers in the trunk are replaced with factorised Gaussian Noisy layers (`NoisyDense`) to drive exploration through parameter noise. Unlike the vanilla agent, exploration does not rely on ϵ-greedy, as noise is injected directly into the weight parameters at training time.

Duelling + NoisyNet SlateQ

The combined variant (`DuelingNoisySlateQNetwork`) uses a NoisyNet trunk with a duelling head, which enhances both exploration and credit assignment. Since exploration relies on parameter noise, ϵ is fixed at zero.

Training Setup

Across all variants:

- **Double Q-learning**: the online network selects the next greedy slate; the target network estimates its value.
- **Loss**: position-weighted Huber loss, optionally masking non-clicked items in vanilla.
- **Gradient Stability**: NaN-safe guards, global-norm clipping, and reward normalisation.
- **Exploration**: ϵ-greedy for vanilla and duelling; parameter-noise-driven for NoisyNet variants.
- **Polyak Updates**: soft target updates with $\tau = 0.003$ or hard sync every 1000 steps.

4.3.3 DQN Agent

The `DQNAgent` learns per-item $Q(s,i)$ without SlateQ's decomposition and treats slate construction as a separate ranking step. The network is a small MLP (256–128–64, ReLU) with a linear head that outputs one value per item. At decision time it forms a slate by taking the top-K items by $Q(s,i)$, optionally multiplied by user–item affinity from the current state.

Training follows Double DQN with a target network: the online net selects the next action, the target net evaluates it. We use Huber loss, reward scaling, and global-norm gradient clipping. Exploration is ϵ-greedy with exponential decay. Targets are updated by Polyak averaging ($\tau = 0.005$) or hard sync every 1000 steps. The loss is applied on clicked items only, which keeps updates aligned with observed feedback while still giving a fair comparison point to SlateQ.

4.3.4 Greedy Agent

The `GreedyAgent` is a non-learning baseline that always recommends the k items with the highest current user–item affinity. It is deterministic and does not explore, which makes it a strong but short-sighted benchmark.

4.3.5 Contextual Bandit

The `CtxBanditAgent` implements a linear contextual bandit (LinUCB-style), learning a simple scoring function from user features. It recommends a slate of top-k items with optional ϵ-greedy exploration. This agent represents short-horizon learning and is widely used in practice for fast, scalable recommendation.

4.3.6 Random Agent

The `RandomAgentShim` provides the simplest baseline by filling the slate with items sampled uniformly at random. Internally, it wraps a lightweight `_RandomPolicy` that samples valid indices within the action specification. Although naive, it serves as a lower bound to check that learning agents are genuinely improving.

4.4 Runtime Execution

The runtime component provides the bridge between the static environment definition in `ECommStory` and the dynamic interaction loop that drives training and evaluation. The `ECommRuntime` class extends the `TFRuntime` from `RecSim NG` and is responsible for stepping the simulation forward, handling agent actions, and returning `TimeStep` objects compatible with `TF-Agents`.

4.4.1 ECommRuntime

The `ECommRuntime` is initialised with a compiled `Network` object from `ECommStory`, and immediately calls the network's `initial_step()` method to create the first environment state. It dynamically infers the number of users, recommender state dimensions, and choice shape from the `user_state` and `rec_state` variables and ensures that the runtime remains compatible with different configuration parameters.

Two key methods define the runtime's behaviour:

- `reset()`: Resets the environment to its initial state and returns a `TimeStep` object marking the start of a new episode.
- `step(action)`: Receives an agent action, constructs the corresponding input dictionary for the network, and advances the simulation by one step. The resulting state is converted into a transition `TimeStep` that includes the new observation, reward, and discount factor.

The runtime also defines `observation_spec()`, `action_spec()`, and `time_step_spec()` methods, which provide the shape and type constraints for agents and replay buffers. Finally, the `trajectory()` method supports rolling out a sequence of steps without agent interaction, which is useful for diagnostics or offline simulations.

4.4.2 Interaction Loop

The main training loop in `main.py` coordinates the interaction between the agent and the environment through the `ECommRuntime`. At each iteration:

1. The agent receives the current observation from the runtime.
2. The agent's `collect_policy` selects an action according to its exploration strategy.
3. The action is passed to the runtime's `step()` method which produces the next `TimeStep`.
4. The experience tuple is stored in the replay buffer for future learning.
5. At defined intervals, the agent is trained using batches sampled from the buffer.

This process repeats for the configured number of steps or episodes, while metrics such as total reward, loss, and ranking scores are logged for later evaluation.

4.5 Metrics and Logging

To monitor agent performance and support later evaluation, the system implements a dedicated metrics module. This module includes ranking quality measures and a logging utility that stores results in both human-readable and machine-parsable formats.

4.5.1 Ranking Metrics

The `ranking_metrics.py` file defines two ranking-based evaluation metrics:

- **NDCG@K** (Normalised Discounted Cumulative Gain): Measures how well the recommended slate matches the ideal ranking for a given user. It is computed by first assigning binary relevance based on whether the ground-truth choice appears in the slate, then discounting the score logarithmically by item position. Scores are normalised by the ideal DCG, which results in values between 0 and 1.
- **Slate MRR** (Mean Reciprocal Rank): Calculates the reciprocal of the rank position of the first correct recommendation in the slate, averaged over all users. This metric reflects how early the correct item appears in the recommendation list.

Both metrics are implemented in TensorFlow to allow seamless integration into the training loop without requiring data export or preprocessing.

4.5.2 Metrics Logger

The `logger.py` file defines the `MetricsLogger` class, which handles writing performance data to disk. Each training run is stored in a uniquely named folder based on a timestamp, containing:

- `metrics.csv`: A comma-separated file suitable for quick inspection and plotting.
- `metrics.jsonl`: A JSON Lines file that preserves full precision and is easy to parse programmatically.

The logger creates file handles and writes a header row to the CSV file upon the first log call. Subsequent calls append rows and flush values immediately, which ensures that progress is recorded even if training is interrupted.

4.5.3 Integration with the Training Loop

During each episode in `main.py`, the runtime provides the agent's selected slates and the ground-truth choices from the environment. These are passed to the ranking metric functions to compute NDCG@K and Slate MRR, while total episode reward and training loss are also tracked. The resulting metrics dictionary is then passed to the `MetricsLogger`, which appends the values to disk for later analysis and visualisation.

4.6 Additional Implementation Details

Beyond the core modules described above, several supporting elements ensure that the system is configurable, reproducible, and easy to maintain.

4.6.1 Configuration Management with Gin

The project uses the `gin-config` library to manage hyperparameters and environment settings. By editing this file, it is possible to run experiments with different settings without modifying the Python source code. Table 4.1 summarises the main hyperparameters used in the experiments. These values were tuned manually through small pilot runs, with the goal of ensuring stability and fairness across agents rather than finding the absolute optimum.

Table 4.1: Key hyperparameters used in experiments.

Component	Key Hyperparameters
Environment	Users = 10, Items = 100, Slate size = 5
ECommUser	Topics = 10, reward mode = sigmoid
	Interest learning rate $\alpha = 0.03$, fatigue $\beta = 0.015$
	Decay $\rho = 0.90$, forgetting rate $\eta = 0.01$
	Diversity bonus $\lambda = 0.20$
Continuation	Logistic coefficients $b_0..b_4$ tuned for novelty momentum
Conversion	Delay probability $p = 0.25$, required exposures = 3
Position bias	Weights = (1.0, 0.75, 0.55, 0.40, 0.30)
Penalties	No-click penalty = 0.05, repeat penalty = 0.15

4.6.2 Development Notes

During development, several modifications were made to ensure compatibility between `RecSim NG` and `TF-Agents`:

- Updating `ECommRuntime` to figure out observation shapes based on the initial state.
- Ensuring `ECommStory` correctly declares dependencies to avoid topological ordering errors in the network graph.
- Normalising reward values in the agent to prevent unstable Q-value growth.

Chapter 5

Evaluation

This chapter presents the evaluation of the implemented recommender system framework and the experimental results obtained. We assess both the overall performance of the agents and the quality of the recommendations, using the metrics defined in Section 3.3. The results are also compared across the different agent configurations when appropriate.

5.1 Evaluation Setup

5.1.1 Hardware and Software Environment

All experiments were conducted on *GitHub Codespaces*, a managed cloud-based development environment. Each codespace was provisioned with:

- vCPU: *4-core virtual CPU (Intel Xeon Platinum 8272CL equivalent)*
- RAM: *8 GB*
- Operating System: *Ubuntu 22.04 LTS (containerised)*
- Python 3.10 with TensorFlow 2.15, TF-Agents, and RecSim NG.

GPU acceleration was not used.

5.1.2 Experiment Parameters

- Number of episodes: 600
- Steps per episode: 200
- Number of users: 10
- Slate size: 5
- Random seeds: fixed for reproducibility

5.2 Results

5.2.1 Learning Curves by Agent

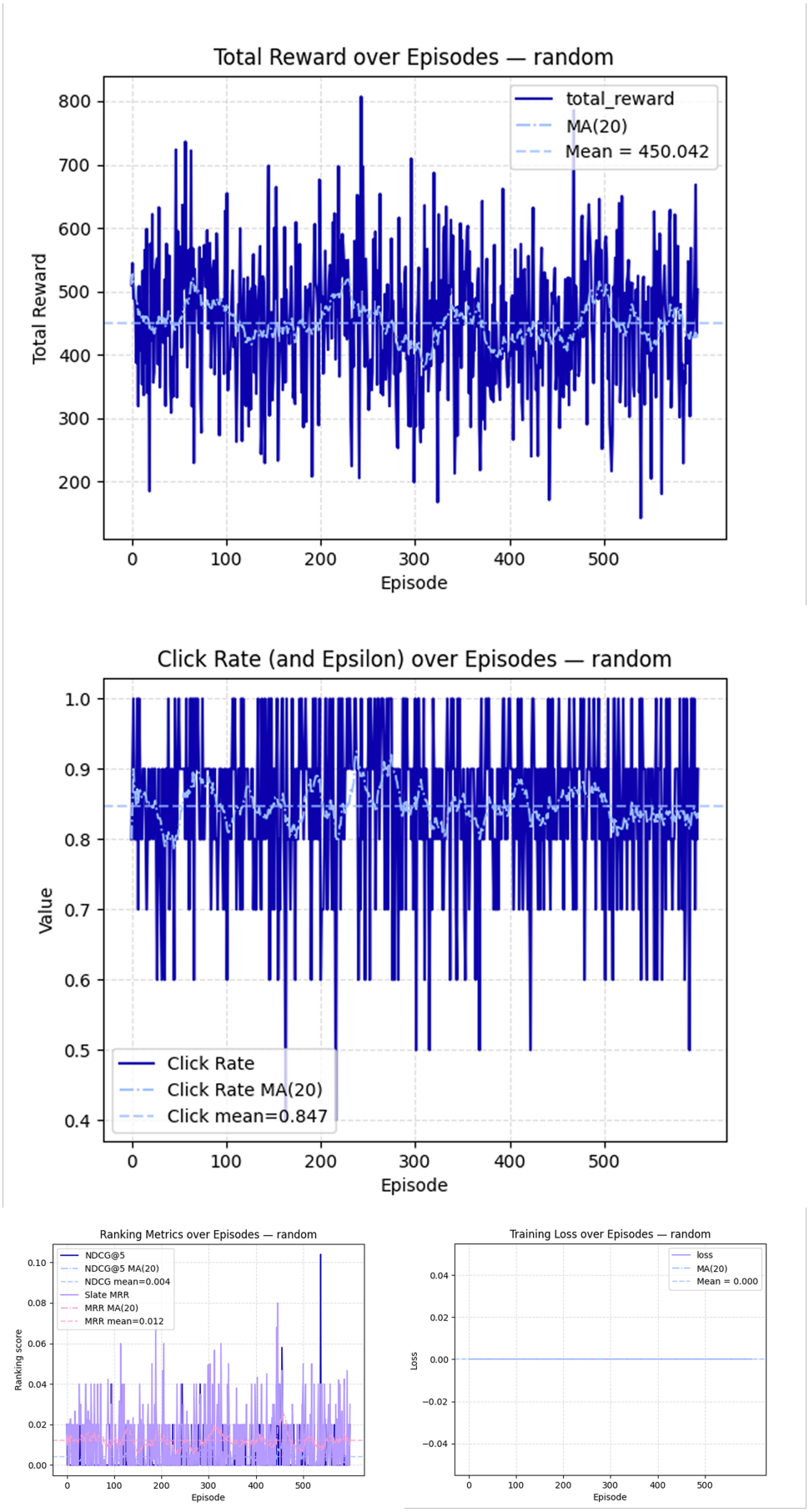

Figure 5.1: Learning curves for the Random agent.

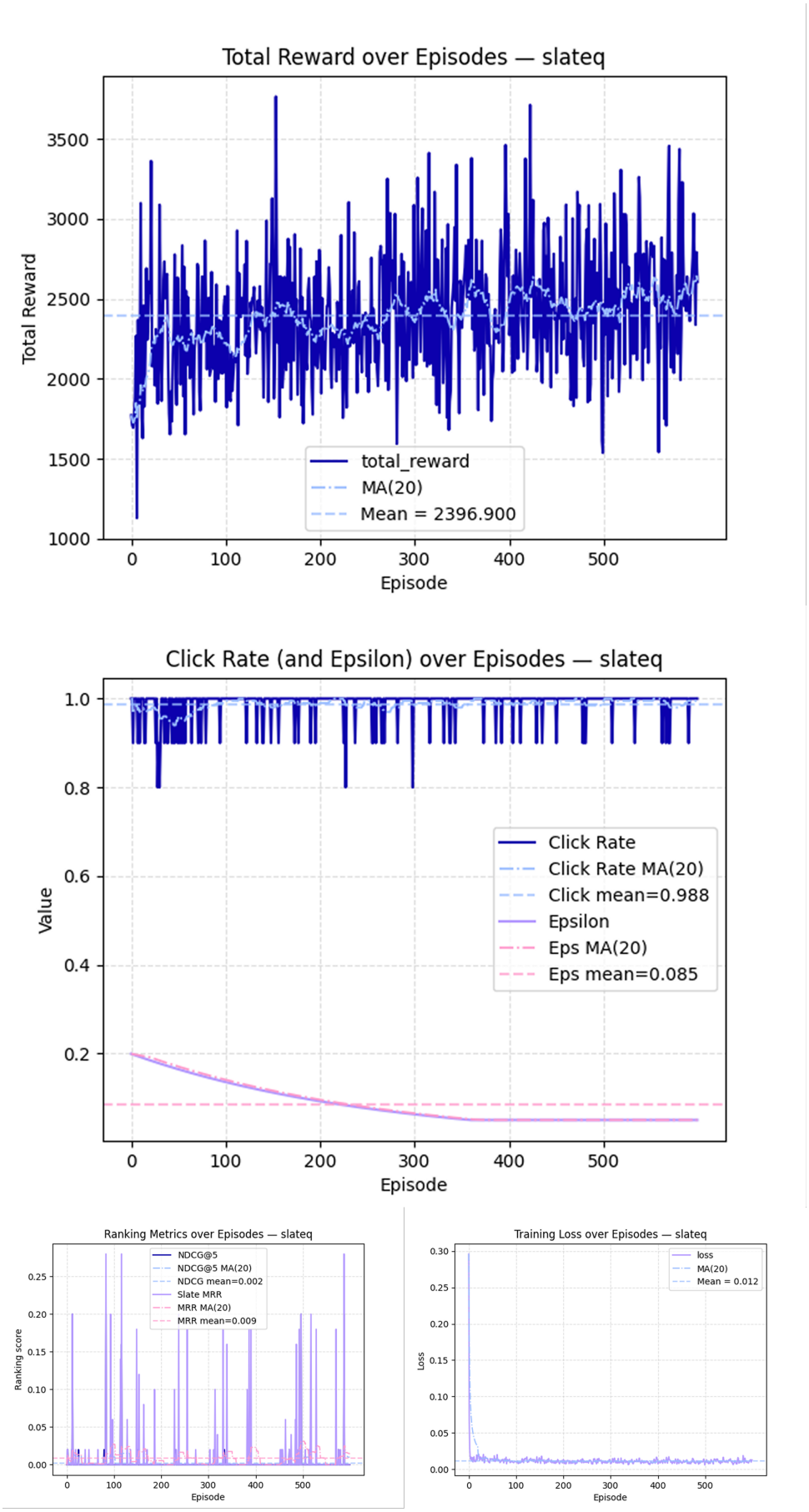

Figure 5.2: Learning curves for the Vanilla SlateQ agent.

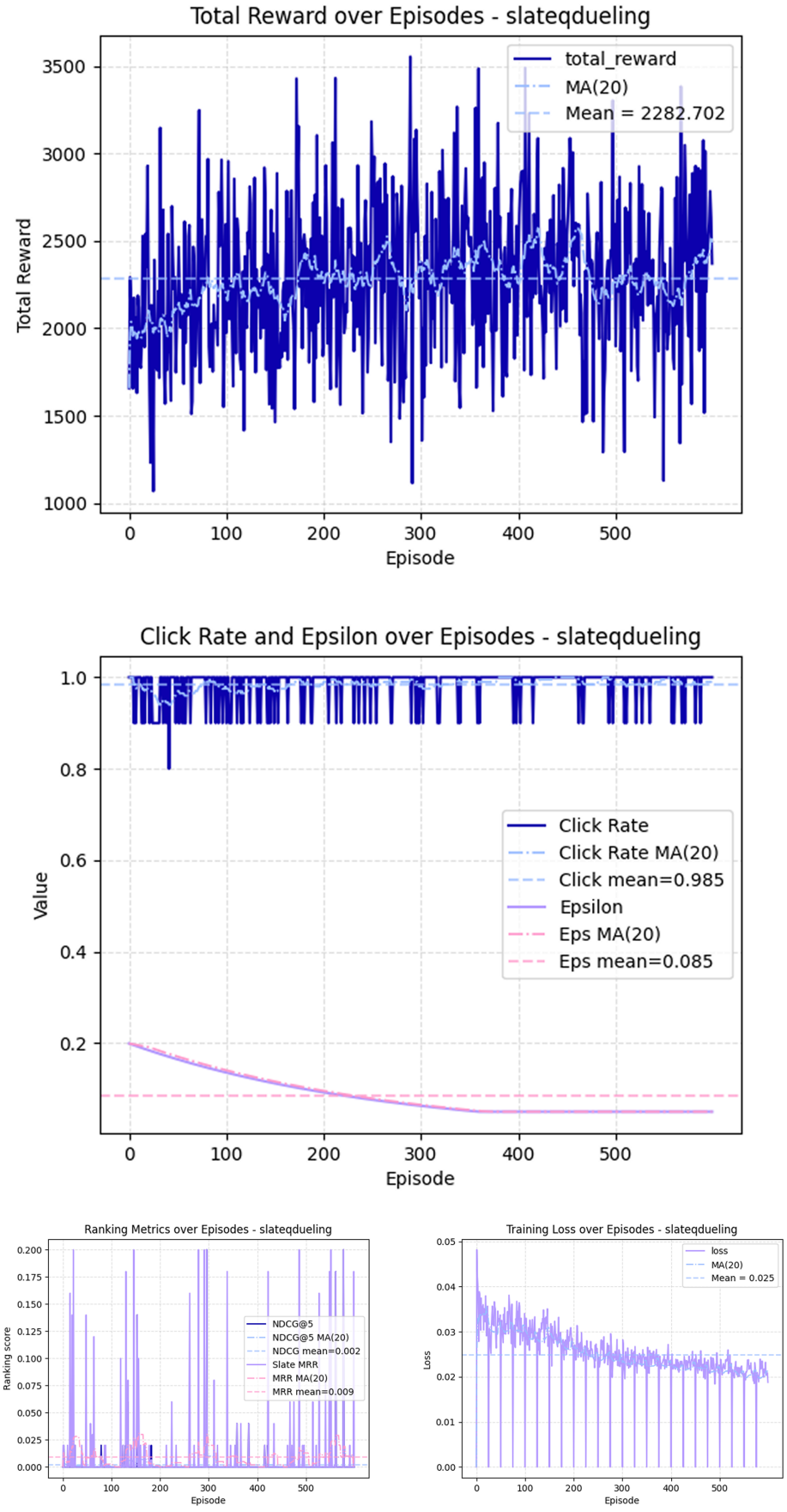

Figure 5.3: Learning curves for the Dueling SlateQ agent.

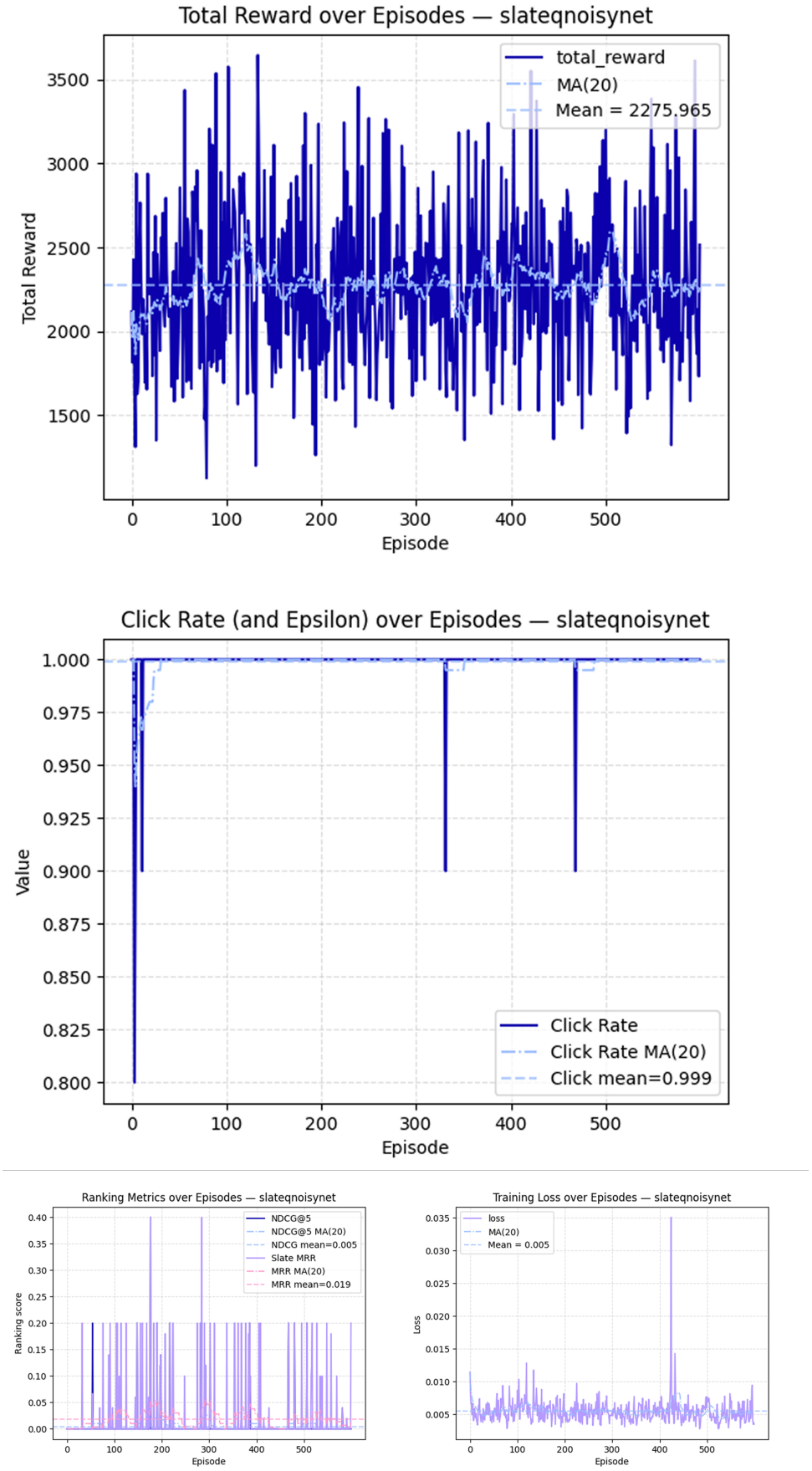

Figure 5.4: Learning curves for the NoisyNet SlateQ agent.

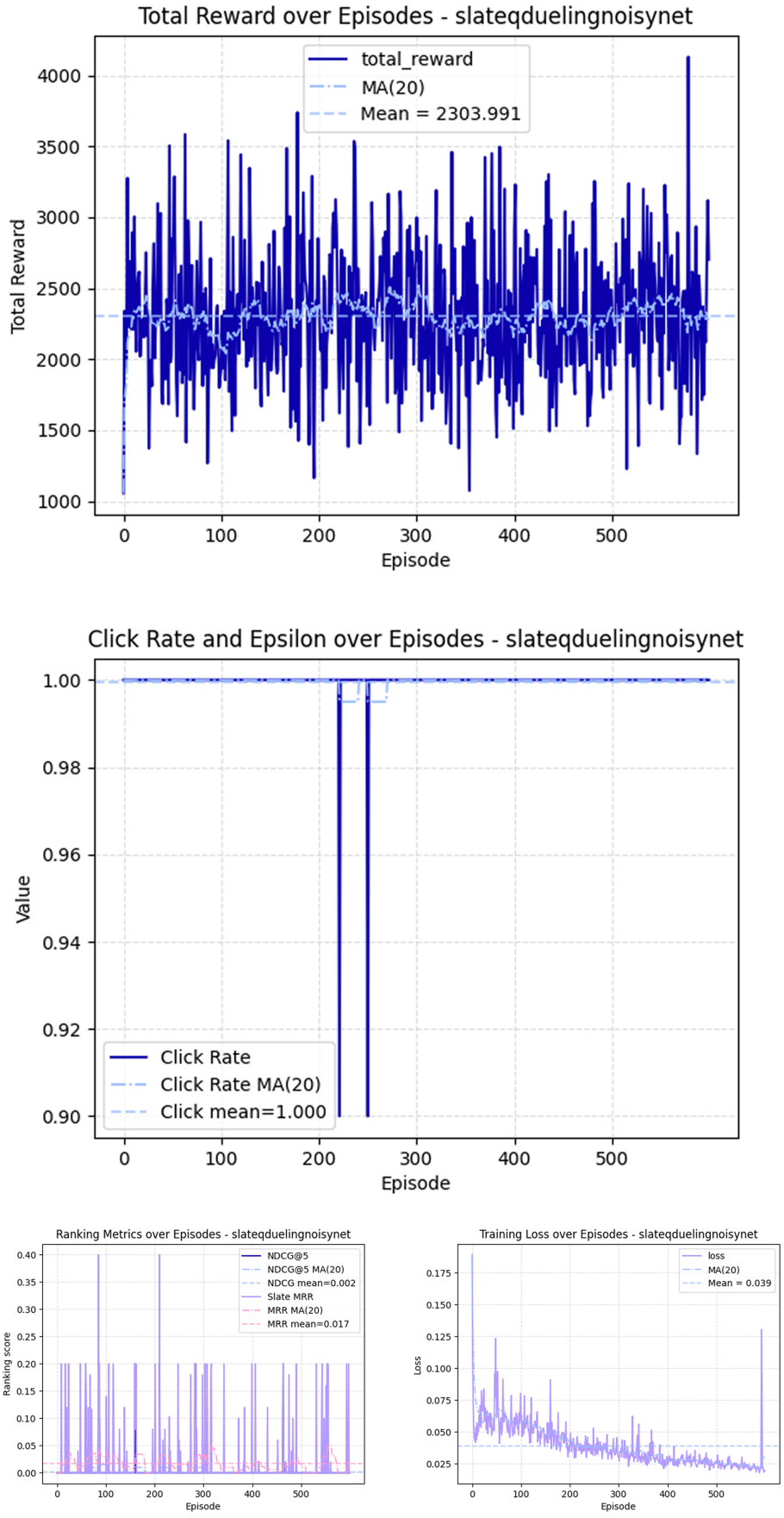

Figure 5.5: Learning curves for the Dueling + NoisyNet SlateQ agent.

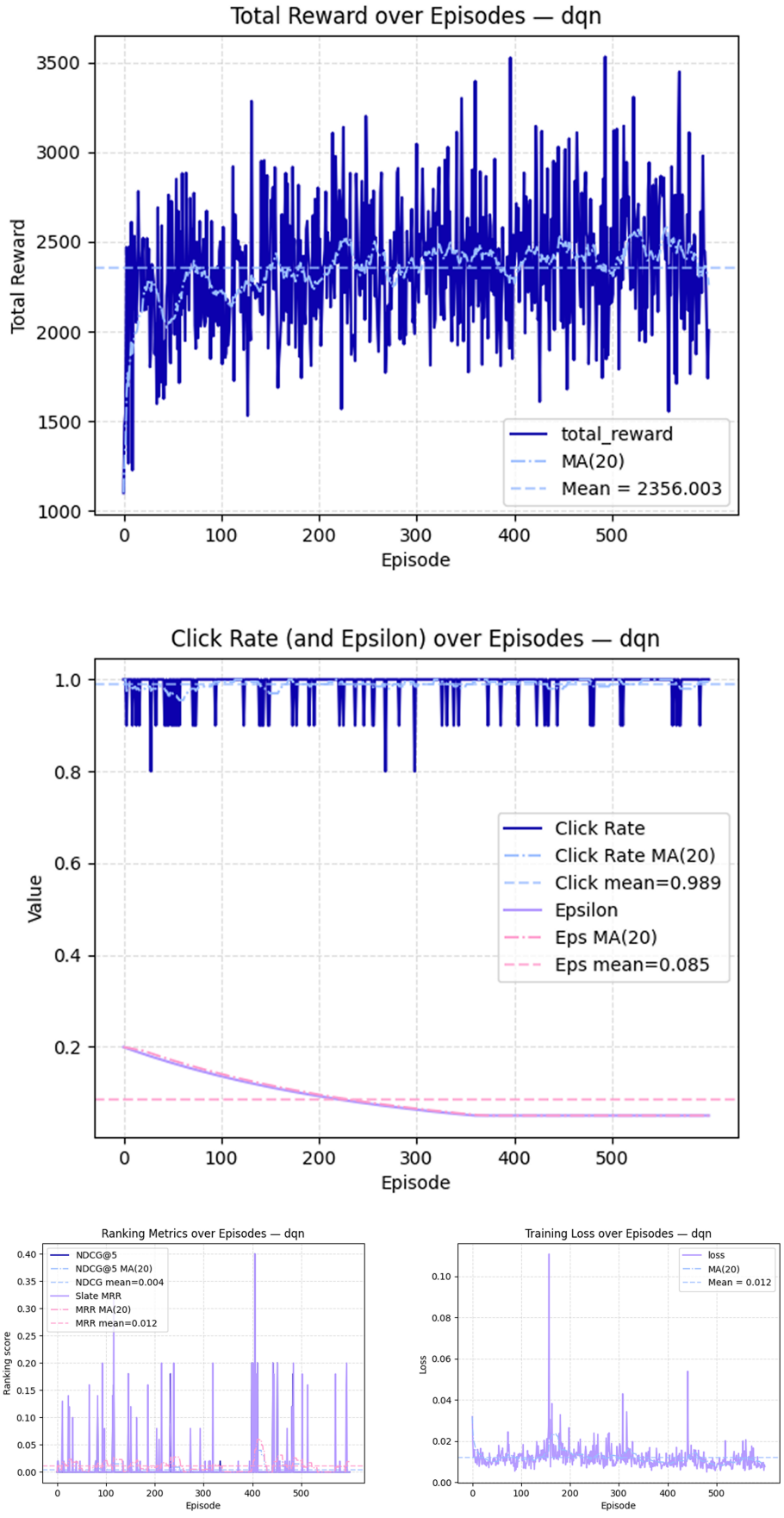

Figure 5.6: Learning curves for the DQN agent.

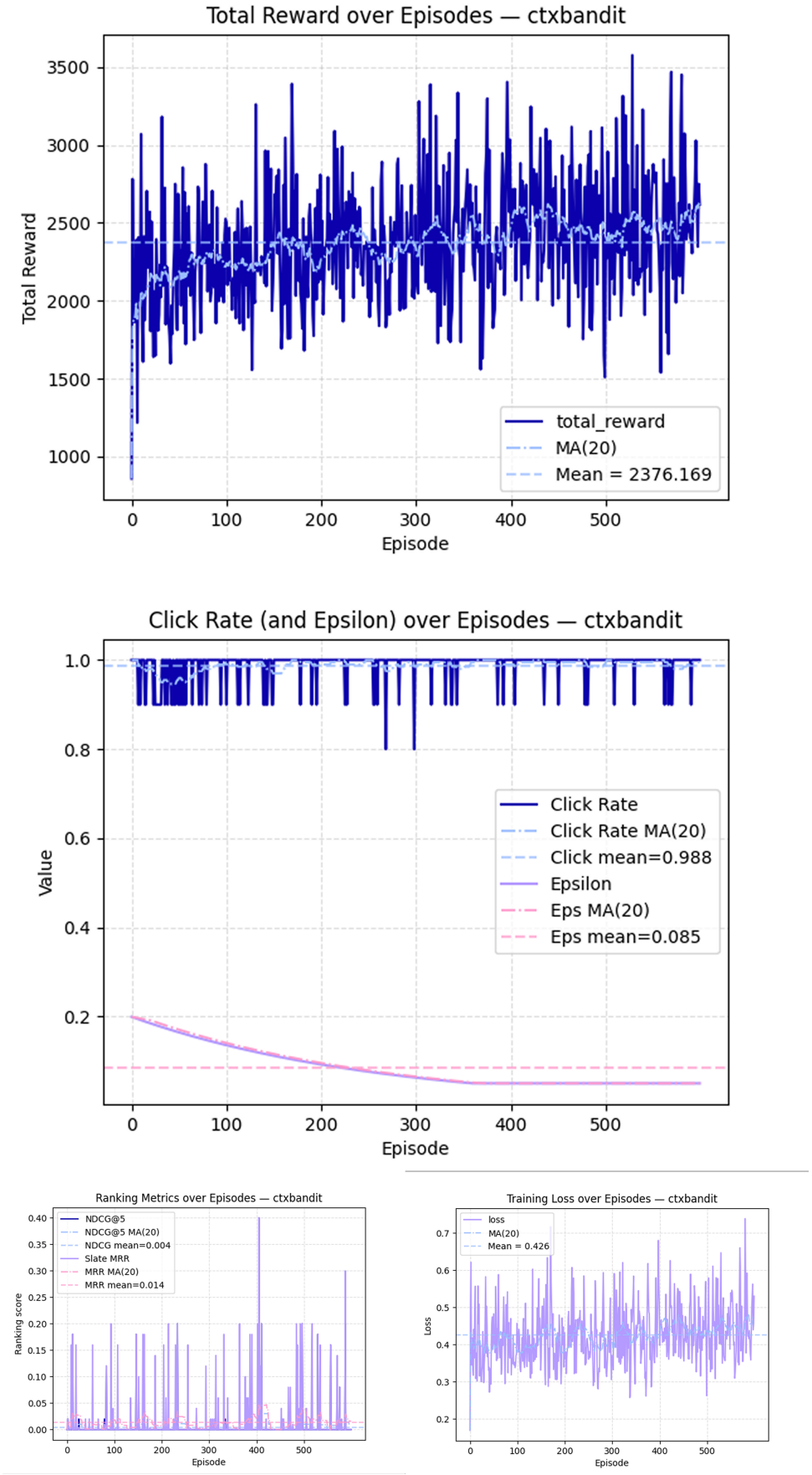

Figure 5.7: Learning curves for the Contextual Bandit agent.

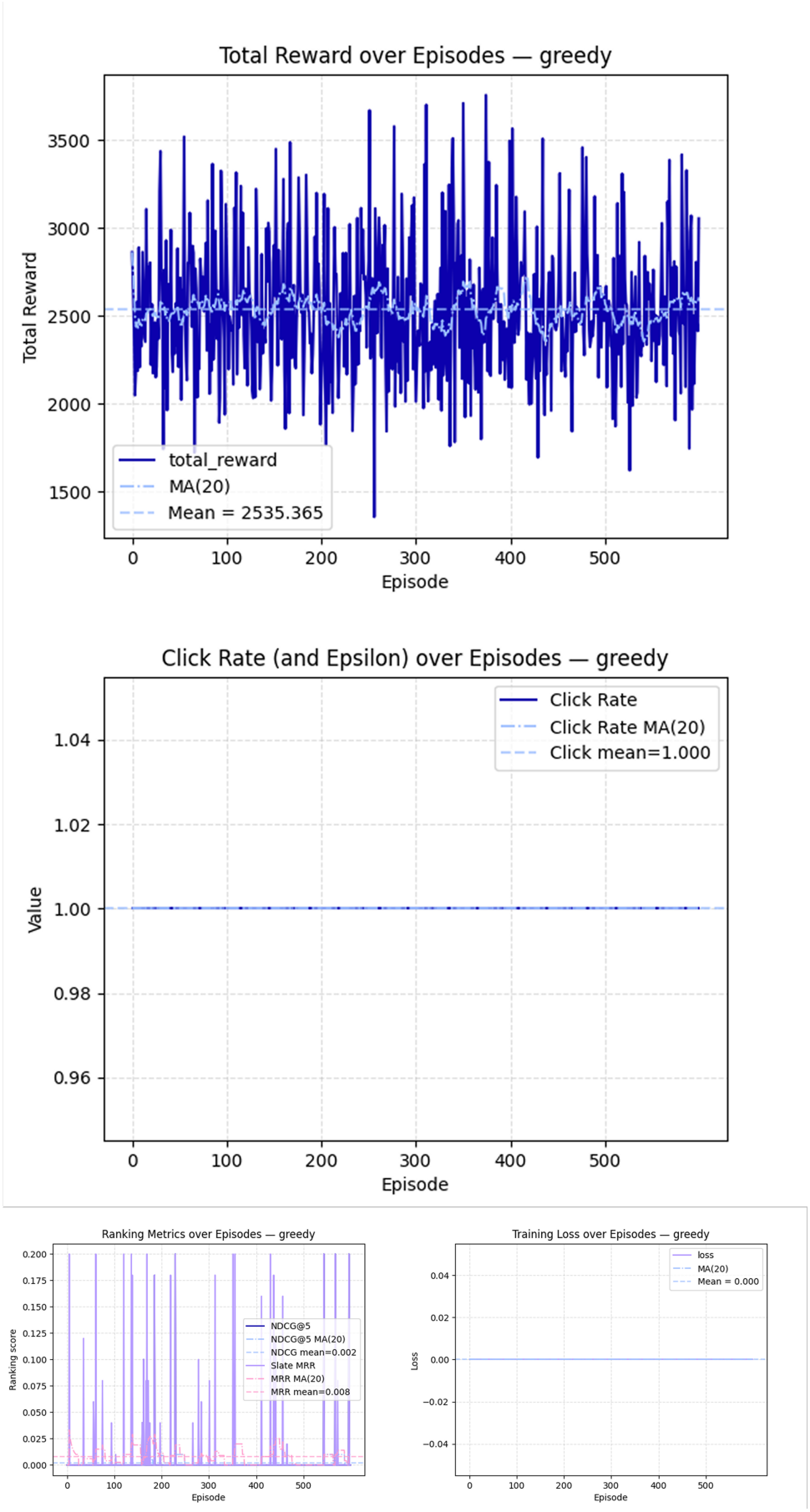

Figure 5.8: Learning curves for the Greedy agent.

5.2.2 Final Performance Summary

Table 5.1: Final averaged performance over the last 100 episodes.

Agent	Total Reward	Loss	NDCG@5	Slate MRR
SlateQ Variants				
SlateQ (Vanilla)	2533.57	0.0108	0.0022	0.0102
SlateQ (Dueling)	2296.25	0.0207	0.0038	0.0130
SlateQ (NoisyNet)	2261.74	0.0051	0.0024	0.0176
SlateQ (Dueling+NoisyNet)	2355.05	0.0250	0.0000	0.0160
Deep RL Baseline				
DQN	2438.74	0.0099	0.0000	0.0090
Simpler Baselines				
Contextual Bandit	2505.54	0.4529	0.0062	0.0148
Greedy	2512.85	0.0000	0.0060	0.0068
Random	446.47	0.0000	0.0041	0.0121

Table 5.2: Final averaged performance across all episodes.

Agent	Total Reward	Loss	NDCG@5	Slate MRR
SlateQ Variants				
SlateQ (Vanilla)	2396.90	0.0116	0.0021	0.0089
SlateQ (Dueling)	2282.70	0.0248	0.0017	0.0092
SlateQ (NoisyNet)	2275.96	0.0055	0.0045	0.0189
SlateQ (Dueling+NoisyNet)	2303.99	0.0388	0.0022	0.0171
Deep RL Baseline				
DQN	2356.00	0.0119	0.0039	0.0117
Simpler Baselines				
Contextual Bandit	2376.17	0.4256	0.0037	0.0138
Greedy	2535.36	0.0000	0.0019	0.0081
Random	450.04	0.0000	0.0041	0.0121

Tables 5.1 and 5.2 report the final averaged results for all agents, both over the last 100 episodes and across the full training run. Several clear patterns emerge when comparing the SlateQ variants, the deep RL baseline, and the simpler baselines.

Within the SlateQ family, the vanilla agent was the most reliable overall and achieved the highest reward in both tables while maintaining stable training. The duelling variant performed worse, with lower reward and higher loss, which indicates that separating value and advantage did not provide a benefit in this environment. NoisyNet produced the strongest Slate MRR, particularly in the overall results where it reached 0.0189, which shows that parameter noise exploration helped surface relevant items earlier in the slate. However, this improvement came at the cost of reward. The combined Dueling+NoisyNet variant sat in between: it recovered

some reward compared to NoisyNet alone and maintained strong MRR (0.0171 overall), but it also recorded the highest loss, which suggests instability. Taken together, these results show that vanilla SlateQ remains the most dependable for cumulative reward, while the variants demonstrate trade-offs between exploration, ranking quality, and stability.

The deep RL baseline, DQN, achieved reasonably high rewards, close to the SlateQ agents, but its ranking performance was weak. In the final episodes, its NDCG@5 dropped to 0.0000, which shows that without SlateQ's decomposition it could not learn effective slate ordering. This highlights the value of SlateQ's structure for handling combinatorial recommendation tasks.

Among the simpler baselines, contextual bandit and greedy were unexpectedly competitive in reward. The bandit's high loss reflects its short-sighted nature, and greedy achieved the highest overall reward at 2535.4, although its ranking quality was weaker. Random, as expected, performed poorly in reward, but its small positive ranking metrics show occasional chance successes. These findings emphasise the importance of benchmarking against strong baselines: although reinforcement learning can improve ranking and long-term modelling, simple strategies can still match or exceed RL in reward under stable conditions.

Chapter 6

Discussion and Future Work

The experiments presented in Chapter 5 give a clear view of how SlateQ and its variants compare to both deep RL and simpler baselines in a slate-based recommendation setting that reflects an e-commerce context. The findings bring out several notable takeaways.

First, the vanilla SlateQ agent stood out as the most reliable in terms of cumulative reward, consistently outperforming the other variants. This suggests that the basic decomposition design of SlateQ already captures the essential dynamics of a slate recommender without extra architectural tweaks. The duelling approach appeared promising in theory but failed to deliver benefits and resulted in higher loss. Assuming the implementation was correct, this suggests that the added complexity is unnecessary in relatively straightforward environments. NoisyNet, by contrast, produced stronger ranking metrics such as Slate MRR, which shows that noise-driven exploration can help agents push relevant items further up the slate. The combined Dueling+NoisyNet approach gave results in the middle: it recovered some reward compared to NoisyNet, maintained solid MRR, but also recorded the highest training loss, which made it less stable. In an e-commerce setting, these results highlight the trade-off between keeping engagement steady and placing attractive products at the top of the slate.

Second, the comparison with DQN highlights the importance of SlateQ's decomposition method. DQN achieved reasonably high rewards, but failed on NDCG@5, particularly in the last 100 episodes where it flatlined at zero. This underlines the limits of per-item approaches in e-commerce, where the ordering of items within a product slate is critical to click-through and conversion rates. Simply maximising per-item Q-values is insufficient when customers engage with whole pages of recommendations.

Third, results from the simpler baselines demonstrate that complex methods are not always necessary. Both contextual bandit and greedy policies achieved reward values comparable to SlateQ. Greedy, in fact, recorded the single highest overall reward, though its ranking quality was weaker. In a real e-commerce setting, this helps explain why heuristic or short-horizon models remain widely used in production: they can deliver strong commercial metrics at relatively low computational cost. However, their weaker performance on ranking quality and long-term modelling suggests they may not sustain user engagement as catalogues grow and customer interests evolve.

There are, however, limitations that need to be acknowledged. The simulated environment used here was deliberately simple, with static items and straightforward user interests. Real-world e-commerce users have far more varied behaviours: they may browse without buying, return

repeatedly over time, and shift preferences in response to promotions, seasons, or trends. Training was limited to single runs with fixed random seeds, so the consistency of results across conditions is not guaranteed. Finally, the reward functions in this project did not explicitly account for diversity, novelty, or business objectives such as revenue or inventory management, all of which are important in practice.

Looking ahead, several directions for future work are worth considering:

- **Multiple runs and variance:** Running the agents several times with different random seeds and then averaging the results would show if the findings are stable and repeatable.
- **Diversity-aware rewards:** Changing the reward to also value variety in the slates could stop the system from always showing the same popular items and make the recommendations more balanced.
- **Richer user models:** Making the user models more realistic, for example by adding repeat buying, seasonal changes, or interest fading over time, would make the experiments closer to real customer behaviour.
- **Scaling experiments:** Trying bigger catalogues and longer slates would test if SlateQ still works well when used at the size of real e-commerce platforms.
- **Hybrid approaches:** Combining simple strategies (such as greedy ranking for popular items) with long-term reinforcement learning could give practical solutions that balance speed with quality.

In summary, the results confirm that SlateQ is a strong candidate for slate-based recommendation, but also that its benefits are most visible when ranking quality matters as much as reward. For e-commerce systems, the key challenge is not just optimising for clicks but managing trade-offs between short-term engagement, long-term satisfaction, and catalogue diversity. Bringing these aspects into future evaluations would help make reinforcement learning methods more practical for deployment in live e-commerce environments.

Chapter 7

Conclusion

This dissertation set out to evaluate SlateQ in the context of slate-based recommendation, with a particular focus on simulated e-commerce environments. The motivation came from the limitations of single-item recommenders and the need to handle the combinatorial complexity of slates while optimising for long-term engagement in online shopping scenarios. In practice, e-commerce platforms present slates, which are sets of products shown together on a page, feed, or carousel. Their quality matters because well-designed slates drive immediate clicks while also supporting long-term customer loyalty (Zhang et al., 2019; Jannach and Adomavicius, 2016).

The project successfully implemented a RecSim NG environment that mimics a simplified e-commerce setting, with users represented by latent interest vectors and items drawn from a static catalogue. Within this setup, multiple agents were designed and evaluated, including SlateQ variants, DQN, contextual bandit, greedy, and random baselines. The evaluation showed that SlateQ vanilla was the most dependable performer in terms of reward, while the duelling variant failed to provide advantages and suffered from higher loss. NoisyNet improved ranking quality, especially Slate MRR, but at the cost of lower reward. The combined Dueling+NoisyNet agent was in the middle: it improved on NoisyNet's reward and continued to deliver strong MRR, but it was also the most unstable. DQN confirmed its limits in slate scenarios by delivering reasonable reward but failing to learn slate ordering, while simple baselines such as greedy and contextual bandit remained surprisingly competitive. These results highlight that while reinforcement learning methods like SlateQ bring value, strong baselines should not be overlooked.

The main contributions of this work are a clean and reproducible implementation of SlateQ and several baselines in RecSim NG, a fair benchmark across different agent types in an e-commerce-inspired setting, and an analysis that highlights both the strengths and the trade-offs of SlateQ in practice. The work also points out some open problems for future work, such as adding more variety to the recommendations, making user models closer to real shopping behaviour, and testing stability by running the agents multiple times.

Overall, this dissertation demonstrates that reinforcement learning, and SlateQ in particular, has real promise for slate-based recommendation in e-commerce contexts. At the same time, it shows that simple strategies can still achieve strong results, and that careful benchmarking is essential before deploying RL in practice.

Word Count

The total word count for Chapters 1–7 (Introduction to Conclusion), excluding references, appendices, tables, and figures, is approximately **8,200** words.

Bibliography

Adomavicius, G. and Tuzhilin, A., 2005. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *Ieee trans. knowl. data eng.*, 17(6), pp.734–749.

Chen, X., Yao, L., McAuley, J., Zhou, G. and Wang, X., 2023. Deep reinforcement learning in recommender systems: A survey and new perspectives. *Knowledge-based systems*, 264, p.110335.

Dulac-Arnold, G., Mankowitz, D.J. and Hester, T., 2019. Reinforcement learning in recommender systems: A complex answer to a simple problem. *Icml 2019 workshop on real world reinforcement learning*.

Ie, E., Hsu, C.W., Mladenov, M., Jain, V., Narvekar, S., Wang, J., Boutilier, C. et al., 2019a. Recsim: A configurable simulation platform for recommender systems. *arxiv preprint arxiv:1909.04847*.

Ie, E., Jain, V., Wang, J., Narvekar, S., Agarwal, R., Wu, R. et al., 2019b. Slateq: A tractable decomposition for reinforcement learning with recommendation slates. *Proceedings of the 28th international joint conference on artificial intelligence (ijcai)*, pp.2592–2599.

Jannach, D. and Adomavicius, G., 2016. Recommendation-based decision making: Models, algorithms, and applications. *Journal of machine learning research* [Online], 17(1), pp.1–36. Available from: `https://jmlr.org/papers/volume17/15-316/15-316.pdf`.

Järvelin, K. and Kekäläinen, J., 2002. Cumulated gain-based evaluation of ir techniques [Online]. *Acm transactions on information systems (tois)*. vol. 20, pp.422–446. Available from: `https://doi.org/10.1145/582415.582418`.

Koren, Y., Bell, R. and Volinsky, C., 2009. Matrix factorization techniques for recommender systems. *Computer*, 42(8), pp.30–37.

Li, L., Chu, W., Langford, J. and Schapire, R.E., 2010a. A contextual-bandit approach to personalized news article recommendation. *Proceedings of the 19th international conference on world wide web*. pp.661–670.

Li, L., Chu, W., Langford, J. and Schapire, R.E., 2010b. A contextual-bandit approach to personalized news article recommendation. *Proceedings of the 19th international conference on world wide web*. pp.661–670.

Ma, Y., Jeunen, O., Mladenov, M., Swaminathan, A., Chen, M., Boutilier, C., Rossetti, M., Paquet, U., Kenthapadi, K., Wang, X. et al., 2021. Recsim ng: Toward principled uncertainty modeling for recommender ecosystems. *arxiv preprint arxiv:2103.08057* [Online]. Available from: `https://arxiv.org/abs/2103.08057`.

Mladenov, M., Wu, H., Kannan, A. and Bastani, O., 2021. Slateq: A tractable decomposition for reinforcement learning with recommendation sets [Online]. *Advances in neural information processing systems*. vol. 34, pp.13557–13569. Available from: `https://proceedings.neurips.cc/paper/2021/hash/4b56a4f41fc2e2c60ddefb8e3d4c75c4-Abstract.html`.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2015a. Human-level control through deep reinforcement learning. *Nature*, 518(7540), pp.529–533.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D., 2015b. Human-level control through deep reinforcement learning. *Nature*, 518, pp.529–533.

Shani, G., Heckerman, D. and Brafman, R.I., 2005. An MDP-based recommender system. *Journal of machine learning research* [Online], 6, pp.1265–1295. Available from: `https://www.jmlr.org/papers/v6/shani05a.html`.

Voorhees, E.M. and Tice, D.M., 1999. The trec-8 question answering track report. *Trec*.

Zhang, S., Yao, L., Sun, A. and Tay, Y., 2019. Deep learning based recommender system: A survey and new perspectives. *Acm computing surveys*, 52(1), pp.5:1–5:38.

Zhao, X., Xia, L., Zhang, L., Ding, Z., Yin, D. and Tang, J., 2018. Deep reinforcement learning for page-wise recommendations. *Proceedings of the 12th acm conference on recommender systems*. pp.95–103.

Appendix A

Source Code

The full implementation developed for this project is openly available on GitHub. It contains the RecSim NG environment, agent implementations, training scripts, and logging tools described in Chapters 3 and 4. The repository can be accessed at:

`https://github.com/mrgw21/SlateQ-RecSys-Research`

This repository provides the complete codebase along with configuration files and instructions for reproducing the experiments and results presented in this dissertation.

In line with submission requirements, a zipped archive of the source code will also be provided together with this dissertation on Moodle.

Appendix B

Project Structure

The codebase follows a modular package layout, with separate directories for agents, environment entities, runtime execution, stories, and metrics:

```
SlateQ-RecSys-Research/
          main.py
          experiments/
                configs/
                    base.gin
          src/
                agents/
                      slateq_*_agent.py  # Vanilla, Dueling, etc.
                      dqn_agent.py
                      ctxbandit_agent.py
                      greedy_agent.py
                      random_agent.py
                entities/
                stories/
                runtimes/
                metrics/
                core/
          logs/
                <agent_name>/run_YYYY_MM_DD_HH_MM/
                    metrics.csv
                    metrics.jsonl
          plots/
              <agent_name>/run_YYYY_MM_DD_HH_MM_*.png
```