A Review of Reinforcement Learning Applications in Ad Policy Optimization for Large-Scale Recommender Systems

Armando Ordorica, Yuri Lawryshyn

University of Toronto, Toronto, Canada

Abstract

Balancing advertisements with organic content in large-scale recommendation engines is a well-known challenge in the industry. While ads are essential for revenue generation in platforms that integrate them alongside organic content, they often underperform in user engagement compared to organic content [**?**]. The key challenges in achieving this balance include ad selection, ranking, and integration with organic content. These involve decisions about whether to insert an ad, which ad to select, and where to place it [**?**]. Traditional methods optimize for the highest probability of ad clickthrough with fixed placements and are typically solved as supervised Machine Learning (ML) problems [**?**]. However, these approaches focus on optimizing single actions, overlooking the downstream impact of sequential user behavior and underutilizing valuable contextual signals. A more dynamic approach that accounts for context and optimizes the long-term value of action sequences, balancing both engagement and revenue, remains an open research area [**?**].

Reinforcement Learning (RL) offers a promising framework to address the need for dynamic, context-aware optimization by formalizing the process of finding the best policy in sequential decision-making under uncertainty [**?**]. However, implementing RL in large-scale recommendation systems faces challenges such as vast state and action spaces, the risk of poorly chosen reward functions, high computational costs [**?**], and the difficulty of interpreting online policy learning [**?**]. This paper surveys existing methodologies and frameworks that utilize RL to enhance recommendation strategies, aiming to guide practitioners and researchers in navigating these challenges. Our analysis reveals that while fully online RL is powerful in theory, in practice, techniques like contextual bandits, offline replay, and surrogate reward modeling often yield more scalable and interpretable solutions for large-scale ad-serving.

Keywords: *Reinforcement Learning, Ad Policy Optimization, Contextual Bandits, Recommendation Systems, Off-Policy Learning, Online Advertising, Deep Reinforcement Learning, Multi-Objective Optimization*

1 Introduction

Platforms such as Google, Meta, TikTok, and Pinterest offer content recommendation systems whose revenue models depend on an effective balance between advertisements and organic content [**?**] [**?**]. This balance is essential to maximize immediate revenue while avoiding long-term issues such as ad fatigue, where users become less responsive to ads due to overexposure, and user churn, where users discontinue using the platform [**?**] [**?**].

Over the past decade, deep learning has emerged as a promising technique that enables recommendation engines to better understand context through embeddings, significantly improving their ability to personalize content [**?**] [**?**]. However, deep learning approaches are often supervised and rely on measurable and observable short-term metrics used as model labels, such as the probability of a click-through or the likelihood of a video being watched in full [**?**]. Methodologies that use myopic objective functions to optimize these directly measurable short-term metrics often overlook the long-term value of a recommendation [**?**]. By focusing on immediate rewards, they may rank content based on short-term engagement metrics while neglecting the entire sequence of user interactions that involve delayed responses. Such approaches can result in suboptimal outcomes, such as promoting clickbait content that compromises long-term revenue and user engagement.

To overcome the challenges of optimizing based solely on immediate objectives, Reinforcement Learning (RL), modeled as a Markov Decision Process (MDP), has emerged as a promising framework [**?**]. RL enables the consideration of long-term value by optimizing policies that maximize cumulative rewards over time, rather than focusing on short-term engagement metrics. In an MDP, actions are selected based on a policy (π), which chooses the action that maximizes the expected cumulative reward from the current state onward. This cumulative reward is typically conceptualized as the expected sum of future rewards, discounted over time using a discount factor. A lower discount factor places more emphasis on immediate rewards, effectively ignoring future outcomes. Conversely, a discount factor approaching one values future rewards more heavily, encouraging the policy to consider long-term benefits [**?**].

In the context of recommendation systems, rewards can be defined to capture both user engagement and revenue objectives. Engagement rewards might include metrics such as time spent on the platform, content views, clicks, saves, shares, or likes [**?**]. Revenue rewards can be tied to ad-related metrics like ad impressions, ad clicks, conversions, or purchases resulting from ad exposure. These different rewards are often combined into a single utility function that reflects the platform's overall objectives [**?**] [**?**] [**?**].

For example, the reward function r_t at time t can be defined as:

$$r_t = \alpha \times \text{Engagement Metric}_t + \beta \times \text{Revenue Metric}_t,$$

where α and β are weighting factors that balance the importance of user engagement and monetization. By appropriately defining the utility function and choosing suitable weights, RL algorithms can optimize policies that enhance user experience while maximizing revenue over the long term [**? ? ? ?**].

Despite the promising theoretical framework that RL offers for optimizing long-term value in recommendation systems, implementing RL in real-world, large-scale platforms remains complex [**?**]. First, in the context of recommendation systems, the state space, which reflects the current context of the user and the environment, and the action space, which captures how the recommender system can be tuned, are both essentially infinite [**?**]. This vastness makes learning and optimization computationally intensive and difficult to discretize into individual actions and states, a critical requirement for policy learning [**?**] .

Second, designing appropriate reward functions is critical, as poorly chosen reward functions can lead to unintended system behaviors that degrade user experience or negatively affect business revenue [**?**]. For example, a reward function that overly prioritizes click-based metrics may drive the system to promote clickbait content, boosting short-term engagement but eroding user trust and satisfaction over time [**?**]. Similarly, a reward function too heavily skewed toward revenue metrics could result in excessive ad load, disrupting the user experience and leading to churn. Striking the right balance in reward function design is essential to achieve both user engagement and monetization objectives without compromising the platform's long-term goals.

Furthermore, interpretability is a major challenge in RL deployment. Understanding the learned policies–the strategies adopted by the RL agent–can be particularly difficult when policies are updated in real-time during user interactions (online policy learning) [**?**] [**?**] [**?**]. This real-time adaptation complicates monitoring and evaluation, making it challenging to understand what is happening during online learning. In extreme cases, this lack of transparency could result in non-compliant or unethical behavior, posing significant risks to both users and the business [**?**].

In light of the challenges outlined above, this paper examines methodologies and frameworks used to implement RL for large-scale recommendation systems, with a particular focus on contexts where ad revenue is central to the business model. Although previous surveys address RL in recommendation systems [**? ?**], they do not specifically explore how RL can balance the competing demands of ad and organic content. This paper fills that gap by analyzing approaches tailored to these unique challenges and bridging the divide between theoretical RL concepts and their real-world applications.

The paper begins with a historical overview of ad selection and ranking systems, tracing the evolution from simple rule-based models and heuristic approaches to tree-based models and modern deep learning techniques. Each of these stages is explored in detail, highlighting their limitations and the motivations for adopting RL as the next paradigm in recommendation systems. Following this contextual history and motivation, the paper introduces key RL components such as the action space, state space, reward functions, and policies, using a straightforward toy example: the frozen lake problem. This example provides a foundation for understanding RL fundamentals, which are then mapped to real-world challenges in online advertising and recommendation systems. Drawing on these analogies, the paper offers a clear and practical framework to help practitioners navigate the implementation of RL and adapt these methodologies to their specific needs.

2 Paper Collection Methodology

To ensure broad coverage, relevant papers were collected from both academic and industry sources. Foundational academic works such as *Reinforcement Learning: An Introduction* [**?**], *Statistical Methods for Recommender Systems* [**?**], and *Artificial Intelligence: A Modern Approach* [**?**] were used to establish key mathematical foundations, including Markov Decision Processes (MDPs), Contextual Bandits, Off-Policy Learning, and Offline Replay.

To reflect the internal perspective of a production-facing ad policy team, a curated list of 10–15 papers referenced within Pinterest's Ad Policy team was also compiled. Examples include *Jointly Learning to Recommend and Advertise* [**?**], *DEAR: Deep Reinforcement Learning for Online Advertising Impressions in Recommender Systems* [**?**], *Ad-load Balancing via Off-Policy Learning* [**?**], and *Practical Bandits* [**?**]. In addition, targeted keyword searches such as "Reinforcement Learning ads [company name]" and "Contextual Bandit [company name]" were conducted for leading companies in the advertising and recommendation space (e.g., Google, Meta, Pinterest, Microsoft, TikTok, YouTube, LinkedIn), surfacing additional papers describing real-world systems and methodologies. To complement those efforts, a separate search was conducted on Google Scholar for relevant review papers. While no surveys were found that specifically focused on RL in ad recommendation systems, related reviews on "contextual bandits", "ad fatigue", and "recommender systems" were used to identify additional references.

Finally, manual crawling of citation networks in cornerstone papers was used to identify additional key contributions. For instance, tracing citations from *Jointly Learning to Recommend and Advertise* [**?**], *DEAR* [**?**], and *Offline Reinforcement Learning* [**?**] led to works such as *Ads Allocation in Feed via Constrained Optimization* [**?**] and *A Contextual Bandit Bake-off* [**?**]. In total, 153 papers were selected and reviewed. Papers were ranked according to three main criteria: influence, to capture the foundational or widely adopted ideas in the field; industry relevance, to ensure that the outcomes were sound, implementable, and actionable; and recency, to reflect the latest methodological developments and deployment practices.

While recent and highly cited work was prioritized, papers introducing novel methodologies were also included regardless of citation count. To contextualize the selected works, this survey builds from the ground up. Each component of the Markov Decision Process, including states, actions, rewards, and policies, is defined and illustrated using a toy example such as the Frozen Lake. This foundation is then extended to practice through a review of how each component is operationalized in industry, along with a discussion of the limitations of those approaches. Drawing on existing methods and lessons from industry, the resulting framework is intended to inform practitioners exploring how RL can be applied in their organizations to develop ad policies that balance monetization with organic engagement. A comparison between the toy example and real-world implementation parallels can be found in Table 1.

3 A Brief History of Ad Selection and Ranking in Recommendation Systems

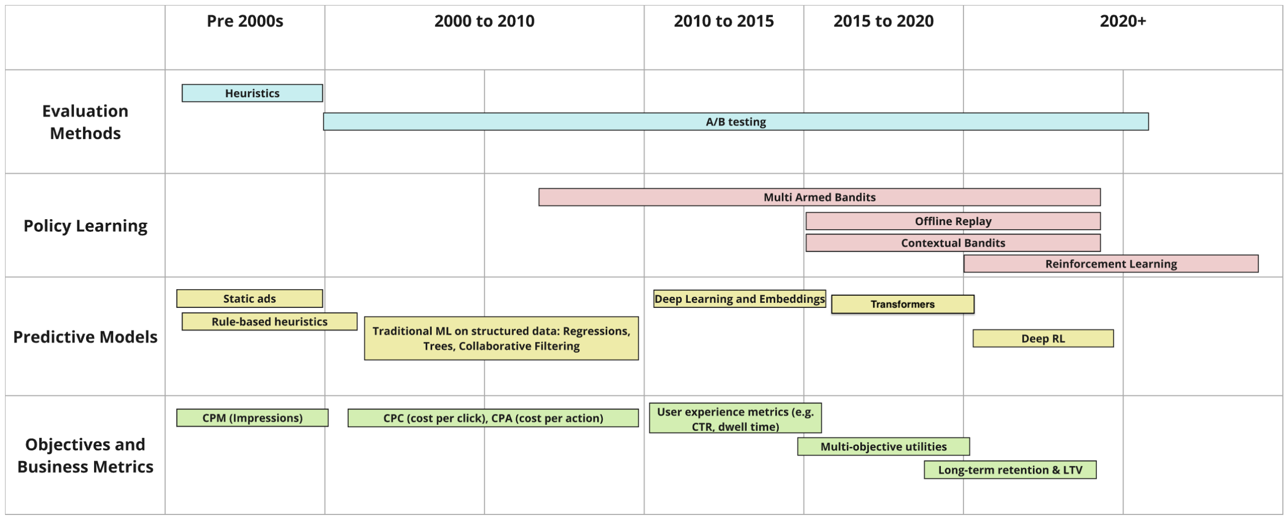


Figure 1: Timeline Showing Initial Adoption and Popularization of Techniques and Methodologies in Online Advertising

Note: The boundaries and lengths of rectangles are approximate. Innovations and transitions between techniques and objectives occurred gradually and continuously, rather than sharply at specific dates.

In online advertising, a good ad policy can result in positive revenue for the company while also boosting user experience. Conversely, a poor ad policy involves showing ads that are irrelevant to the user's current intent, overloading the feed with excessive ad density, and failing to align content with the user's interests. Irrelevant ad targeting can lead to ad fatigue, a state where users become desensitized or frustrated by excessive or poorly targeted ads, resulting in decreased engagement with both the ads and the platform as a whole [**?**].

This section traces the evolution of advertising recommendation algorithms, culminating in the adoption of RL to overcome the limitations of earlier approaches. It begins by outlining how early online advertising primarily mirrored offline methods through banner ads and other static formats. It continues by explaining how static online advertising evolved over time to incorporate rule-based models and more sophisticated heuristic techniques. The discussion then transitions to the integration of machine learning methods, including regression-based models, tree-based algorithms, and collaborative filtering, followed by the emergence of deep learning and embedding-based approaches that effectively capture the nonlinear interactions between users and content. Finally, RL is introduced as a promising solution to address the shortcomings of preceding methodologies. This evolution is visually represented in Table 1, which shows the initial adoption and popularization of techniques and methodologies in online advertising.

3.1 From Rule-Based Advertising to Traditional Machine Learning

Around the early 2000s, online advertising largely mirrored offline practices, heavily relying on non-targeted formats such as banner ads [?]. These generic advertising methods conflicted with the objective of search engines of efficiently directing users to relevant content [**?**]. As a marginal improvement on nontargeted ad formats like banner ads, advertisers began to utilize simple `if-then` logic with predefined rules to select and display ads based on basic criteria like user demographics. While rule-based systems were straightforward to implement, their lack of flexibility and personalization became evident as ad fatigue, defined as reduced user engagement due to repeated exposure to the same advertisement, persisted [**?**]. This issue underscored the necessity of not only leveraging user signals but also optimizing ad load tuning parameters, such as frequency and placement, to maintain user interest and maximize click-through rates [**?**]. For instance, researchers at LinkedIn introduced concepts like top slot - highest eligible position for an ad within the feed - and minimum gap - the required distance between consecutive ads to prevent oversaturation - as ad load tuning parameters [**?**]. Their research indicated that users are more likely to click on ads that are spaced farther apart [**?**].

The evolution of ad pricing, from paying for impressions to paying for user actions, created pressure within the online advertising industry to improve relevance and performance-based metrics [?]. Sponsored search, introduced by GoTo (later Overture), allowed advertisers to bid on specific keywords, directly linking ads to user queries and addressing the limitations of generic advertising [**?**]. Foundational pricing models, such as Cost per Mille (CPM), charged advertisers per 1,000 impressions and closely mirrored offline practices [**?**]. Yahoo!, through the acquisition of Overture [**?**], later introduced Cost per Click (CPC) [**?**], a performance-based pricing model that shifted costs to reflect measurable user engagement. This evolution continued with the introduction of Cost per Action (CPA), which tied fees to specific user actions beyond clicks, such as purchases or sign-ups. These changes incentivized the advertising industry to innovate, leading to the development of more relevance-focused strategies. These foundational pricing models remain integral to the utility functions that underpin contemporary ranking algorithms [**?**]. As advertisers increasingly demanded performance-based pricing, the ad tech industry was compelled to innovate within a competitive yet lucrative market, addressing the limitations of rule-based and heuristic models through the adoption of machine learning techniques to enable more dynamic and data-driven approaches [**?**].

The next set of innovations following heuristics and performance-based targeting was led by machine learning (ML) and A/B testing [?]. From 2010 to 2015, ad selection techniques evolved significantly with the adoption of machine learning models such as Logistic Regression, Decision Trees, and Gradient Boosting Machines [**?**]. By leveraging historical data and diverse user behavior signals, these models enabled more accurate predictions of user engagement [**?**]. In parallel, A/B testing allowed practitioners to systematically compare models and features, ensuring that any new implementation delivered measurable performance gains [**?**]. Despite these successes, traditional tabular models struggled to capture nuanced user intent, particularly when processing more complex data such as images or text. In the early 2010s, deep learning gained momentum with breakthroughs like AlexNet [**?**], which demonstrated the power of neural networks to handle unstructured high-dimensional data. This ability to capture complex, nonlinear interactions dramatically improved ad recommendations, especially through the use of embeddings, which are dense vector representations that map words or images into high-dimensional spaces. Metrics like cosine similarity further refined personalization by aligning recommendations with user interests [**?**]. Embedding-based models, particularly two-tower architectures, presented a major leap forward in ad recommendation systems by effectively capturing intricate relationships between users and ads. These models utilize two neural networks: one dedicated to encoding user information and the other to encoding ad (item) information. The outputs from these networks are then combined to predict user engagement with ads [?]. This approach not only boosted accuracy but also scaled effectively. Despite the significant improvements that embedding-based models introduced in terms of predictive accuracy and scalability of ad recommendation systems, the practical challenges of balancing multiple objectives, such as maximizing revenue, mitigating ad fatigue, and sustaining long-term user engagement remained unresolved.

Multi-Armed Bandits (MABs) emerged as a powerful method for online advertising around the 2010s to minimize user exposure to suboptimal ad configurations, effectively tackling the explore–exploit tradeoff, which consists of finding the optimal balance between choosing new ad policies with unknown performance and showing older ad policies with known positive performance [?] [?]. MABs are a class of algorithms for dynamic decision-making under uncertainty, originally developed in mid-20th-century probability theory [?]. While MABs had been widely applied in areas such as clinical trials [?] [?], they only gained significant traction in online advertising around the 2010s [?].

MABs allow platforms to reduce unnecessary exposure to a suboptimal treatment by continuously updating the probability distribution of which treatment is most likely to yield the highest reward [?]. In clinical trials, for instance, a bandit algorithm reduces unnecessary exposure to suboptimal treatments by updating its reward estimates in real time: it allocates more patients to the better-performing treatment (exploitation) while still testing others that could be better (exploration) [?][?]. In this setting, the “reward” might be a measurable improvement in patient health or recovery time [?]. The same principle applies to online advertising, where “treatments” translate into specific ad configurations, defined by parameters such as displacement cost, reserve price, ad load, and ad placement. “Rewards” in online advertising typically represent user engagement metrics such as clicks or conversions [?]. Advertising platforms should continuously “explore” new treatments to test new ads, formats, and targeting strategies to avoid stagnation, while simultaneously “exploit” existing high-performing treatments to ensure near-term revenue targets are met [?].

In contrast to traditional A/B tests, where there are randomized treatments assigned for a predetermined amount of time, MABs dynamically learn which configuration yields the highest reward for each user at a particular time. This approach reduces the time users are exposed to suboptimal ad configurations, thereby mitigating losses in engagement and long-term revenue [?]. Although this dynamic learning framework effectively tackles the explore–exploit tradeoff, defining the precise set of rewards and objectives to optimize remains an open challenge, particularly when advertisers must juggle multiple goals such as revenue and long-term user satisfaction [?].

To address this need for clear, multi-objective definitions, such as balancing user satisfaction and revenue, utility functions have been employed in online advertising since the early 2000s [?]. Rooted in microeconomic theory dating back to the mid-20th century, utility functions are mathematical constructs that quantify preferences and the net value of conflicting objectives [?] [?]. Introduced to formalize decision-making under uncertainty, they model and quantify consumer preferences, enabling advertisers and publishers to encode multiple objectives, such as revenue, user satisfaction, and long-term platform health, into a single or composite utility metric [?]. By integrating utility functions into MAB-based frameworks, advertisers can systematically guide the exploration–exploitation process toward a unified reward signal that balances short-term performance (e.g., clicks, impressions, or conversions) with long-term considerations (e.g., minimizing ad fatigue or maintaining user trust). This approach allows for principled trade-offs, optimizing for immediate gains while safeguarding long-term engagement.

3.2 Advances in Deep Learning: Attention Models, Transformers, and Contextual Bandits

While embedding-based models greatly enhanced the predictive power of recommendation systems, capturing the full complexity of user sequences, especially over long time spans, remained a challenge for sequential deep learning methods such as Long Short-Term Memory networks (LSTMs) and Gated Recurrent Units (GRUs) [?] [?]. Traditional recurrent networks (e.g., LSTMs or GRUs) process sequences step by step, typically encoding an entire sequence into a single fixed-size vector [?]. This design poses several challenges. Because Recurrent Neural Networks (RNNs) operate strictly in a sequential manner, they cannot process multiple parts of the sequence in parallel, creating bottlenecks and risking the loss or blurring of details, especially those from earlier steps in long sequences [?]. Moreover, standard RNNs inherently prioritize recent inputs as information is passed along, which attenuates earlier signals and leads to the well-known vanishing gradient problem in lengthy sequences and fails to effectively capture long-term dependencies [?][?].

To address training inefficiencies in RNNs and the difficulty of capturing long-term dependencies, attention mechanisms allow direct access to any position in the sequence, enabling the model to focus

on relevant features in the sequence regardless of their temporal distance [?]. By computing pairwise interactions across all positions in the sequence, Transformers overcome the sequential bottlenecks of RNNs and enable parallel computation [?]. This mechanism also allows the model to incorporate contextual signals, such as user demographics, session-level metadata, or item attributes, at any point in the sequence, rather than strictly depending on a running hidden state [?]. The ability to attend to any part of the input at any time allows Transformers to model long-range dependencies with far greater efficiency and accuracy than RNN-based architectures [?]. This paradigm shift led to the development of the Transformer architecture, which entirely replaces recurrence with self-attention, fundamentally redefining sequence modeling [?].

The Transformer's parallel attention architecture is a natural fit for modern hardware. Since self-attention and feed-forward layers are implemented with large matrix operations, the model makes efficient use of Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs), which excel at parallel computation [?]. Unlike RNNs that left many GPU cores underutilized due to their step-by-step nature, Transformers keep the hardware busy by processing many elements concurrently. This yields significantly shorter training times compared to earlier recurrent models. Moreover, Transformers scale up seamlessly: training can be distributed across multiple GPUs, and larger batches or longer sequences can be handled without a drop in throughput [?]. Engineers can simply add more GPU workers to accommodate bigger models or more data, achieving near-linear speedups in many cases [?]. The ability to train deep models efficiently at scale has unlocked previously impractical projects, and Transformer-based architectures (e.g. BERT, GPT) are now standard building blocks in modern AI systems [?] [?]. One of the key innovations of the Transformer architecture is its multi-head attention mechanism [?]. Instead of relying on a single attention distribution, multiple attention "heads" are trained in parallel, each with its own set of learned parameters. This design allows the model to capture a range of relationships or patterns within the same sequence [?].In practice, each head can focus on different aspects. One head might emphasize local dependencies in the user's interaction history, while another one might highlight more global, long-range correlations [?].

In an advertising context, multi-head attention naturally accommodates multi-objective learning [?]. For example, one head could be tuned to prioritize short-term engagement metrics (e.g., click-through rate or CTR) by attending to recent user activities and relevant ad features, while another head could emphasize longer-term outcomes, such as mitigating ad fatigue or preserving user satisfaction [**?**] [**?**]. Each head learns a distinct set of attention weights, enabling the model to isolate and capture unique signals that serve different goals [**?**]. The outputs from these multiple heads are then aggregated, typically concatenated and linearly transformed, into a single representation or score, providing a unified but multi-faceted embedding of user–ad interactions [**?**] [**?**]. However, determining the appropriate attention heads and their optimal weighting remains a complex challenge [**?**][**?**] [**?**]. The optimal weights may vary not only across different users and content types but also over time [**?**] [**?**]. Additionally, shifting business objectives, engineering constraints, and competing optimization objectives further complicate the design process [**?**] [**?**] [**?**].

Selecting the right heads in Multi-Task Attention Models and determining their optimal weighting is not a deterministic process [?] [?]. It typically requires iterative experimentation, involving both offline analysis (e.g., hyperparameter tuning, head ablation studies) and online testing (A/B experiments) [**?**] [**?**]. This process is particularly challenging for new users and new content, where slow adaptation can result in suboptimal personalization and early churn [**? ?**].

To address difficulties related to policy optimization - like weight tuning and head selection - *Offline Replay* is often employed to find optimal policies in large parameter search spaces [?] [?] [?]. This technique typically begins with data collection via A/B/n experiments, where different policies are randomly assigned to users [**?**] [**?**]. Afterward, Offline Replay simulates the effects of various ad policies across different user segments and estimates the potential rewards based on logged interactions [**?**]. In this context, “reward” refers to the performance metric we aim to maximize (i.e. clicks, likes, shares, revenue) [**?**]. However, in many cases, the true optimal treatment is not explicitly tested [**?**]. Here, importance sampling corrects for differences between the observed behavior policy and the target policy, improving estimation accuracy [**?**] [**?**]. Based on the collected data on random assignments between policies and users typically involved in Offline Replay, one can use different policy learning methods to map the optimal policies from user ids, content types, and other contextual signals for the optimal weights at any point in time [**?**].

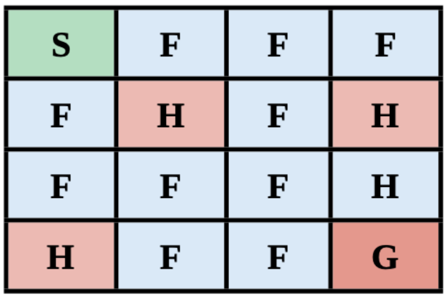
Offline replay often assumes that user preferences and item availability do not change significantly during data collection. However, in real ad systems, these factors can shift, causing offline data to become stale or biased [**?**] [**?**]. Contextual bandits address this limitation by leveraging continuous feedback from live user interactions, preventing the policy from being locked into a static offline dataset [**?**]. Contextual bandits can be thought of as a special case of RL, where an action selection does not influence future states [**?**] [**?**]. Unlike full RL, which models long-term state transitions, contextual bandits assume that each decision is independent, optimizing for immediate rewards rather than a multi-step trajectory [**?**] [**?**]. This makes them particularly well-suited for recommender systems, where ranking decisions can be treated as independent events [**?**] [**?**]. Additionally, contextual bandits naturally balance exploration and exploitation, allowing them to adapt dynamically to user behavior in real time [**?**]. However, a practical limitation of contextual bandits is that they select actions from a fixed set defined at deployment time, such as preconfigured ad policies or treatment arms [**?**]. As a result, the policy's performance is inherently constrained by the quality and granularity of the available states and actions at the time of inference [**?**].

Although contextual bandits inherently focus on short-term rewards, they can be tuned to approximate long-term objectives by encoding historical or delayed outcomes into the feature space or the immediate reward function [?] [?]. For instance, if an attention head emphasizing ad engagement indirectly correlates with better retention, a contextual bandit can increase its weight over time, provided that retention signals are reflected in the contextual features or the reward proxy [**?**]. However, full RL remains more appropriate for scenarios demanding explicit multi-step optimization, where actions taken now significantly shape future user states [**?**]. Contextual bandits thus offer a practical trade-off, providing continuous adaptation and efficient online learning without the computational complexity of fully modeling multi-step state transitions [**?**] [**?**].

Despite the practical advantages that contextual bandits offer for real-time experimentation and short-horizon reward maximization, they remain limited in scenarios where user states evolve over repeated interactions [?]. Modern advertising platforms often contend with extended user sessions and multiple recurring visits, demanding an approach capable of modeling multi-step feedback loops. RL naturally extends these capabilities by incorporating stateful dynamics and cumulative rewards, capturing phenomena like ad fatigue [**?**] and long-term retention [**?**]. Recent industrial deployments underscore RL's growing importance in ads: ByteDance uses RL-based systems [**? ?**] to balance immediate revenue with session-level engagement in short-video feeds, Meta (Facebook) has leveraged policy-gradient methods for sequencing notifications and ad placements over evolving user states [?], and companies like Google are experimenting with RL-driven ad scheduling that accounts for shifting user contexts at scale [?]. These efforts demonstrate how RL can transcend the myopic focus of bandits to address long-horizon outcomes, multi-objective trade-offs, and dynamic user behaviors. In the sections that follow, we explore how these RL paradigms are integrated into large-scale advertising stacks, focusing on core algorithms, design decisions, and lessons learned from real-world implementations.

4 The RL formulation of the Ad Recommendation Problem

In RL, the **GridWorld problem** is a foundational example of a Markov Decision Process (MDP), where an **agent** interacts with a **deterministic environment** by performing **discrete actions**, such as moving up, down, left, or right, to navigate through a grid with the objective of reaching a goal, while potentially avoiding obstacles or hazards. A more complex variant is the **FrozenLake problem**, which features a **stochastic environment** where movements are uncertain due to a slippery surface (F) as shown in Figure 2 [?]. Here, **actions do not always lead to the intended next state**, adding complexity by requiring policies that effectively manage uncertainty. In this paper, **we use the FrozenLake problem as a representative framework for stochastic environments and leverage this template to describe the ad optimization problem as an RL task**. Specifically, we will explore components of the underlying MDP, such as states, actions, and rewards, and investigate key RL concepts including policy evaluation, value iteration, and policy iteration. The grid is typically represented as shown in Figure 2:



S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	G

Figure 2: Frozen Lake Environment Grid

4.1 The Agent and its Objective

While toy examples like GridWorld [?] and Frozen Lake [?] provide well-defined, relatively simple environments where RL algorithms can be easily tested, transitioning these insights to real-world recommendation systems is far more complex due to factors like vast state–action spaces and delayed rewards, and environment observability [? ?]. **Observability refers to the degree to which an agent can directly perceive the complete state of the environment** [?]. In fully observable settings (such as the FrozenLake), the agent sees the true state at every step, enabling straightforward decision-making [?]. However, real-world environments, including **recommendation systems, are often partially observable [?]**; the agent must infer the underlying state from incomplete or noisy signals, increasing both the computational complexity of learning and the uncertainty of policy outcomes [?]. In the FrozenLake environment, the **agent is a decision-maker that observes its surroundings and chooses actions**. It learns to navigate the grid to reach the goal (G) from the start position (S) while avoiding holes (H) and navigating slippery surfaces (F). By analogy, in ad policy optimization, **the agent is the algorithm responsible for serving ads**, which must balance two primary **objectives**: maximizing revenue while maintaining (or ideally enhancing) user engagement [? ?]. Ultimately, both the FrozenLake agent and the ad-serving algorithm share the same high-level goal: to maximize long-term rewards through strategic actions grounded in observed states.

The use of a **reward signal** to formalize the idea of a goal is a distinctive feature of RL [?]. In an MDP, the reward is a scalar value the agent receives after taking an action in a given state and transitioning to a new state, serving as immediate feedback on the benefit or cost of the action [?]. This feedback allows the agent to learn a policy that maximizes the cumulative reward over time [?]. In simple RL scenarios, rewards are typically straightforward. For instance, the agent might receive -1 per move (to encourage shorter paths), +100 for reaching the goal (G), and -100 if it falls into a hole (H), ending the episode. **However, in real-world recommendation systems, reward structures can be far more nuanced. It is often unclear which user behaviors warrant the highest reward or how to weigh different signals** [?]. In online advertising, for example, low-intent actions such as clicks may be valued less than high-intent actions like saves or purchases. Moreover, deciding how to balance these varied signals in a single reward function can be both subjective and complex, especially when multiple objectives (e.g., revenue, user satisfaction, and content diversity) must be optimized [?].

Ad recommendation engines often need to **balance multiple objectives**, such as maximizing user engagement while also ensuring ad revenue [**? ? ?**]. Long-term engagement and revenue are metrics that are particularly difficult to measure directly, especially in randomized controlled experiments that typically span only a couple of weeks [**?**]. The delay in outcomes (e.g., retention signals may take months to manifest) and the often subtle impact of a single intervention can make it nearly impossible to detect meaningful changes within such short time frames [**?**]. As a result, proxy metrics are commonly used. For instance, click-through rate (CTR) and ad impressions serve as approximate measures of revenue, while session duration, saves, and shares act as proxies for longer-term engagement [**?**]. However, these simpler proxies can fail to capture the system's true long-term objectives, occasionally leading to suboptimal policies [**?**]. A high CTR may reflect short-term engagement, yet it can ultimately induce ad fatigue or frustrate users if the ads are misaligned with their actual interests [**? ?**]. Table 3 summarizes some of the most commonly used proxies in the industry across dimensions of revenue, engagement, and ad fatigue.

Utility functions are mathematical constructs used to estimate the net value of conflicting objectives [?]. A utility function is a **composite reward function** that allows the agent to integrate various rewards or metrics from multiple objectives using a unified currency, facilitating more nuanced decision-making across competing priorities [**?**]. The ability to compare two utilities, like those of revenue and engagement normalized to a common scale, is critical for any principled blending of organic items and ads [**?**]. Most utility functions are linear though this linearity is not a requirement for how the individual utility of each component is calculated [**? ?**].

A **canonical utility function** at the policy layer to *blend* ads and organic content can be represented as follows [**? ? ?**]:

$$U(x) = \alpha \times \text{Revenue} + \beta \times \text{Engagement} - \gamma \times \text{Ad Fatigue} \tag{1}$$

where α, β, and γ are weights that determine the relative importance of each factor. To compute the utilities of "Revenue", "Engagement", and "Ad Fatigue", typically many signals feed into each component. These signals include, but are not limited to, the ones shown in Table 3. In a bandit or RL context, $U(x)$ serves as the reward signal guiding policy updates. The process of tuning the weights of α, β, and γ is often done through experimentation, as the right balance between engagement and revenue can shift depending on the context. For instance, ByteDance uses a multi-objective reward framework in their recommendation systems, where different reward functions are designed for organic content and ads, combining them to guide the overall recommendation strategy [**?**].

In some cases, a very simple unified function can suffice over a complex utility function. Researchers at Alibaba introduced the concept of Click Yield, which is defined as the ratio of the total number of clicks on all items (ads + organic results) to the total number of impressions on a page [**?**]. Click Yield provides a holistic evaluation of page performance, accounting for the interactive effects between ads and organic content, and helping to mitigate the bias that arises from examining CTR in isolation. Their optimization task maximizes total revenue while ensuring that the Click Yield does not fall below a certain threshold (T) [**?**]. A comparison of utility function formulations is shown in Table 4.

4.2 Action Space Representation

In RL, environments are modeled as a Markov Decision Process (MDP). An MDP is defined by $(\mathcal{S}, \mathcal{A}, P, R)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $P(\cdot \mid s, a)$ is the transition probability distribution, and $R(s, a)$ is the reward function. At each timestep t, the agent observes a state $s_t \in \mathcal{S}$ and selects an action $a_t \in \mathcal{A}$. Both action and state sets may be finite, infinite, discrete, or continuous.

In RL, an **action** $a \in \mathcal{A}$ refers to **a decision or intervention taken by the agent based on its current state to influence the environment** [**?**]. The action space $\mathcal{A}$ represents the complete set of all possible actions that the agent can choose from at any given decision point [**?**]. For example, in the FrozenLake environment, **the possible actions for the agent are *up, down, left* and *right*** [**?**]. **Because this set of possible actions is countable, the action space for the FrozenLake is said to be *discrete* [?].** By contrast, in an *advertisement recommendation* setting, the action space $\mathcal{A}$ might consist of all possible combinations of ads, their rankings, placements, and formats [**?**]. The objective of this set of actions in ad recommendation is to deliver the right set of ads at the right time in order to maximize platform revenue, while maintaining or improving user engagement and satisfaction [**?**]. Although this action space is discrete and countable, the number of possible combinations grows exponentially with the number of candidate ads and available slots. As a result, it becomes computationally infeasible to enumerate or evaluate each action individually, and must instead be treated using methods designed for high-dimensional discrete spaces [**?**].

An ideal action space should be *expressive* enough to capture all relevant degrees of freedom the agent needs to optimize its behavior, but it should avoid unnecessary complexity [? ?]. The action space described above in the FrozenLake example satisfies these criteria, as the four cardinal movements (*up, down, left,* and *right*) are enough to navigate to any state in the grid while remaining **orthogonal**, thus providing both sufficient **coverage and simplicity** [**?**]. In contrast, the ad recommendation action space is inherently high-dimensional and entangled [**?**]. The agent must decide *what* to show (a discrete choice over candidate ads), *where* to place it (which may be modeled as a discrete slot or a continuous position), and in *what order* (often determined by continuous relevance scores), often under tight latency constraints and user personalization requirements [**?**]. The action space in ad recommendation systems must be expressive enough to encompass all relevant choices for optimal ad delivery while ensuring no combinatorial explosion in complexity [**?**]. Moreover, ensuring that each dimension remains sufficiently *orthogonal* can be nontrivial, given potential interactions between ad content, positioning, and user context [**?**]. If the action space is too coarse, the system risks underfitting, serving generic policies that fail to personalize effectively or miss monetization opportunities [**?**]. On the other hand, if it is too fine, the system may overfit to noise, struggle to generalize, or incur prohibitive computational costs [**?**]. Practical systems often resort to action abstraction or parameterization (e.g., slate-based policies, ad load knobs, or policy buckets) to reduce complexity at the cost of optimality [**? ? ?**].

The independence of these action dimensions facilitates more modular policy learning [? ?]. By allowing a degree of independence for each action dimension, the agent achieves modular policy learning that scales linearly (instead of exponentially) with the number of action degrees-of-freedom [**?**]. In addition, actions should ideally correspond to **semantically meaningful choices** to facilitate policy interpretability and validation [**? ?**]. In the case of the FrozenLake problem, the action space is intuitive and human-interpretable, making it straightforward to validate learned policies [**? ?**]. For instance, an RL practitioner can manually inspect and reason about the learned policy (e.g. "the agent plans to move *down* then *right* to reach the goal") to verify that each action choice aligns with domain logic [**?**]. While the action space in ad recommendation systems is far more complex, structuring it around semantically meaningful levers (e.g., “increase ad load for high-intent users" or "downrank low-quality creatives") enables engineers and product teams to reason about the system's behavior [**? ?**]. If a learned policy increases ad load for a particular user segment, analysts can trace this decision back to upstream signals, such as historical engagement patterns or commercial intent scores [**? ?**]. Policies composed of interpretable levers enable more transparent debugging, offline validation, and policy review, which is crucial for ensuring that the deployed strategy aligns with business intuition, avoids learning spurious correlations, and mitigates the risk of unintended outcomes [**? ? ? ?**].

Moreover, an action space should be consistent across episodes [?]. That is, taking the same action in the same state should yield statistically predictable outcomes [**?**]. **The state space in the FrozenLake is stochastic [?]**, meaning that taking the same action does not always result in the same next state. For example, selecting the action "move right" might result in the agent moving right, up, or down, as governed by the transition probabilities defined by $P(s' \mid s, a)$ [**?**]. Despite the stochasticity in the environment, **the outcomes observed should converge to the expected values described by the transition probability distribution over a sufficiently large number of episodes [?]**. A similar principle applies in ad recommendation. Although user behavior and auction dynamics introduce stochasticity, **taking the same ad-serving action, such as presenting a specific slate of ads to a user with a given context, should yield statistically predictable engagement or monetization outcomes**. That is, over many episodes, **the distribution of observed rewards (e.g., clicks, conversions, revenue) should converge to the expected value associated with the given state-action pair [? ?]**. This consistency is foundational for policy learning, as it ensures that repeated exposure to similar decision contexts enables accurate estimation of long-term value, despite noisy observations [**?**].

The action space can be thought of as levers that can be used to tune ad policy. Typical **levers in treatments in ad policies** include the following elements [**?**]:

- **Ad Load.** Also referred to as the spacing ads or frequency of ads is also a key component in inducing ad load fatigue. Placing ads too closely together may lead to user fatigue and reduced click probabilities [**? ?**].

- **Ad Quality.** This refers to intrinsic features of the ad, such as the design format, product type, accompanying text, and the quality of the landing page, all of which directly influence user engagement. [**? ?**].

- **Ad Placement.** Ads placed earlier in the feed are more likely to be clicked, though their relevance to preceding content also plays a significant role [**? ? ?**].
- **Ad Relevance.** Ads that align more closely with user preferences and user intent are more likely to drive engagement. For example, an ad on a hotel might have a higher probability if shown adjacent to results of search queries related to vacations [**? ?**].

4.3 State Space Representation

In RL, a *state* $s \in \mathcal{S}$ represents all the relevant information from past agent-environment interactions necessary for the agent to make optimal decisions to maximize the expected cumulative reward [**?**]. Ideally, the state space is Markovian, meaning the future is conditionally independent of the past given the current state [**? ?**]. If the agent's policy or value estimates rely on omitted or unobserved information from the past, the state representation violates the Markov property and the performance of the learned policy may degrade as a result [**? ?**]. In the FrozenLake example, the state space consists of the agent's current grid position as the future outcomes of any action depend only on this position, and not on how the agent arrived there [**?**]. In this example, because future transitions depend only on the current state and action, and not on prior history, the FrozenLake Environment satisfies the *Markov Property* [**? ?**].

In contrast, the environments where **ad recommendation systems** operate are often **partially observable, high-dimensional, and non-stationary** [**? ?**]. To construct a state representation that reasonably approximates the Markov property in the face of partial observability and complex user dynamics, ad recommendation systems often combine tabular features with learned embeddings [**? ?**]. The tabular signals tend to include static attributes (e.g. demographics), contextual metadata (e.g. device type, time of the day), and behavioral history (e.g. recent clicks, impressions, or dwell time) [**? ?**]. An embedding is a learned, dense vector that encodes complex or high-cardinality inputs, such as user profiles, ads, or behavioral sequences, into a lower-dimensional space [**?**]. Compared to static tabular features like frequency counts or recency buckets, embeddings capture richer structure, including sequence effects and latent similarity. This makes them more expressive and task-relevant, enabling ad recommendation systems to personalize content and ads more effectively [**? ?**].

Embeddings may consist of two components: an offline (stationary) component and an online (non-stationary) component [?]. **Offline embeddings** represent **stationary or slow-changing attributes**, such as a user's demographics (e.g., gender, country) or long-term preferences, and fixed item properties (e.g., brand or product category) [? ?]. These embeddings are usually computed in batch mode due to their **higher computational complexity, larger size, and relatively static nature** [? ?]. **Online embeddings**, conversely, address non-stationary and rapidly evolving aspects such as recent user behaviors, trending content, or immediate interests [? ?]. These embeddings are **generally smaller, computed in near-real-time to meet latency requirements** [?]. This hybrid approach combines stable and dynamic embeddings, balancing relevance with computational efficiency [? ?]. Furthermore, this allows the recommendation system to remain stable under distribution shifts, which is of paramount importance, especially in environments with rapidly evolving user preferences, content inventories, and behavioral patterns [? ?].

To effectively integrate diverse embeddings (user, item, interaction) and their offline and online components, embeddings of varying dimensions are typically concatenated and then projected via learned linear layers into a shared dimension [? ? ?]. Generally, zero-padding or truncation is avoided, as these methods can introduce noise or cause information loss [?].

User embeddings are learned vector representations designed to capture user preferences and behaviors, enabling ad recommendation systems to personalize content effectively at scale [? ?]. These embeddings compress complex, high-dimensional user data, such as past interactions and demographic information, into compact vectors utilized by recommendation models [? ? ?]. User embeddings can be learned using a combination of supervised, unsupervised, and self-supervised approaches [?]. In the case of supervised learning, embeddings are optimized to predict explicit labels such as clicks, conversions, or purchases [? ?]. To exploit information in more implicit signals, self-supervised methods can be used with objectives such as next item interaction or item co-occurrence [? ?]. Many practical implementations tend to combine both supervised and self-supervised methods [? ?]. For example, embeddings may initially be learned through self-supervised tasks (e.g., predicting next clicks) and then fine-tuned on supervised tasks (CTR, conversions). A well-documented example is that of YouTube, where despite the availability of explicit feedback mechanisms (thumbs up/down, in-product surveys, etc.), they turn to implicit signals, where a user completing a video is a positive example, given that the relative prevalence of implicit signals is much higher than explicit signals, allowing the models to learn more effectively [**?**].

Item embeddings are designed to capture intrinsic item characteristics [**?**]. These embeddings compress complex, high-dimensional item data, such as product descriptions, categories, brands, and historical interaction patterns [**? ?**]. Similar to user embeddings, item embeddings can be learned through supervised, unsupervised, and self-supervised approaches [**? ?**]. In supervised learning scenarios, embeddings are optimized to predict explicit labels, such as clicks, conversions, or purchases associated with the items [**? ?**]. To leverage more implicit information, self-supervised methods may focus on objectives like predicting co-occurrence patterns or sequential interactions (e.g., item-to-item relationships) [**? ?**].

While user and item embeddings provide essential representations of individual entities, it is through interaction embeddings, which capture the joint dynamics between users and items, that the system truly learns user preferences and can surface content that is contextually relevant and personalized [**?**]. Being able to **effectively represent user-item interactions** is at the core of personalization for recommendation systems [**?**]. Collaborative Filtering (CF) was one of the earliest popular methods to represent user-item interactions [**? ?**]. CF predicted user preferences by analyzing explicit interactions (e.g., ratings) or implicit behavioral signals (e.g., clicks, views), using metrics like cosine similarity or Pearson correlation [**?**]. However, CF struggled with sparse datasets, as many user-item pairs lacked interactions, limiting recommendation accuracy [**?**]. Additionally, CF's direct interaction-based approach limited its ability to capture deeper latent or implicit relationships between users and items [**?**].

Matrix Factorization (MF), popularized during the Netflix Prize competition (2006–2009), addressed some of CF's key limitations [**?**]. The classic approach is to factorize the (sparse) interaction matrix into the product of a user-factor matrix and an item-factor matrix [**?**]. MF learns these latent (hidden) user and item factors by optimizing embeddings to reconstruct observed interactions, enabling it to capture deeper relational structures that CF missed [**?**]. MF also significantly improved storage efficiency [**?**]. For instance, a CF matrix with 1 million users and 1 million items would contain 1 trillion (mostly sparse) entries. In contrast, MF represents these interactions using two dense embedding matrices totaling approximately 128 million entries (assuming an embedding dimension of k=64) [**?**].

As companies began logging large-scale user interaction data in the late 2000s and early 2010s, Amazon, Spotify, and Netflix turned to Matrix Factorization (MF) as a simple yet effective way to harness that data for scalable personalization [**? ?**]. However, while MF improved predictions for existing users/items with limited interactions, it did not solve all limitations of CF [**?**]. Particularly, MF still struggled with the cold-start problem, as completely new users or items without interaction history cannot be effectively represented in the embedding space [**?**]. Additionally, MF assumes linear interactions between latent factors, limiting its ability to capture more complex, nonlinear user behaviors [**?**]. Neural embeddings emerged in the 2010s to address some of these limitations [**?**]. Unlike CF and MF, neural embeddings naturally capture complex, nonlinear interactions between users and items [**?**]. They also effectively incorporate content-based and contextual metadata, significantly improving the handling of cold-start issues [**? ? ?**]. Two-tower models, notably introduced by Google for YouTube's recommendation systems, leverage deep neural networks for scalable candidate retrieval and generation [**?**]. A two-tower model consists of two distinct neural network components: one tower encodes user-specific features, and the other one encodes item-specific features, projecting them into a shared latent embedding space [**?**]. This shared embedding space enables efficient similarity computation, facilitating large-scale real-time recommendations [**?**]. Today, companies like YouTube use two-tower models for tasks such as next-video prediction, and Meta uses them for personalized ad ranking [**? ? ?**].

Following the widespread adoption of two-tower neural networks, the need to explicitly capture temporal dynamics emerged, motivating the rise of sequence-aware embeddings [**?**]. Users' preferences evolve continuously, and earlier methods struggled to effectively capture these dynamics [**?**]. Earlier embedding methods, such as Matrix Factorization and two-tower models, primarily represented user-item relationships statically, without adequately capturing evolving user preferences or the order of interactions [**? ?**]. Sequence-aware embeddings thus emerged as an important advancement [**?**]. Initially, RNNs, particularly GRUs, were introduced to model sequential user behaviors [**?**]. GRUs could effectively capture short-term temporal dependencies by processing sequences step-by-step, making them well-suited for real-time recommendation tasks [**? ?**]. However, these recurrent architectures faced several key limitations: they were prone to the vanishing gradient problem, where information from earlier interactions diminished during training, making it challenging to capture long-range dependencies [**?**]. They also exhibited a recency bias, disproportionately emphasizing recent interactions and overlooking longer-term user interests [**?**]. Moreover, their sequential nature resulted in slow training times, limiting their scalability [**?**]. To address these limitations, Transformers, originally introduced by Vaswani et al. in 2017, became widely adopted for recommendation tasks in the late 2010s and early 2020s [**?**]. Transformers leveraged self-attention mechanisms, allowing the model to simultaneously process all interactions in a sequence and dynamically attend to the most informative interactions—such as pivotal clicks, conversions, or moments of high engagement [**?**]. This adaptive attention mechanism effectively mitigated the vanishing gradient problem, robustly captured both short-term and long-range dependencies, and reduced recency bias by emphasizing interactions based on relevance rather than mere chronological proximity [**? ?**].

Transformers enable parallel processing over sequence elements, significantly accelerating training and making them well-suited for large-scale recommendation systems with extensive historical data [**? ?**]. Industry models like Pinterest's PinnerFormer (2020) and ByteDance's DEAR (2021) have leveraged this capability to effectively model evolving user behavior and serve contextually relevant recommendations [**? ?**].

4.4 Policy Learning

A policy is a core component of any RL system [**?**]. It encodes not only the logic that determines which action the agent should take in a given state to maximize long-term cumulative rewards, but also how much to explore versus exploit [**? ?**]. Exploration is necessary for the agent to discover the value of unfamiliar actions, which may lead to better strategies [**?**]. However, this must be balanced with exploitation, which involves choosing actions that are already known to yield high rewards [**?**]. Without a policy, the agent has no systematic way to select actions or improve its behavior based on feedback from the environment [**?**].

Formally, a policy is denoted as $\pi(a|s)$, representing the probability of selecting action a given state s [**? ?**]. In the context of the FrozenLake example, a policy is the mapping of grid positions (e.g. the states) to the optimal moves *up, down, left, right* (e.g the actions) in order to maximize cumulative reward [**?**]. The FrozenLake setting also provides intuition for the explore–exploit tradeoff [**?**]. If the agent only explores, it never settles on a reliable strategy and may keep falling into holes [**?**]. If it only exploits, it might stick to a suboptimal path and miss shorter or safer routes [**?**]. Balancing both is essential to finding the optimal policy [**?**].

The degree of randomness in a policy is an important consideration [?]. A policy can be either stochastic or deterministic [?]. In a ***stochastic*** **policy**, $\pi(a|s)$ defines a probability distribution over the action space: $\sum_a \pi(a|s) = 1$ and $0 \leq \pi(a|s) \leq 1$ for all a [?]. In a ***deterministic*** **policy**, the agent always selects the same action in a given state, with $\pi(a|s) = 1$ for exactly one action and 0 for all others [?]. A deterministic policy can be expressed as $\pi(s) = a$, where, in the context of the FrozenLake example, s is the agent's current position in the grid and a is the action selected from the set {*up*, *down*, *left*, *right*} [?]. For example, if $\pi((2,1)) = $ *down*, it means that when the agent is in row 2, column 1 of the grid, it should move down to maximize the expected return [?].

It is important to distinguish between the stochasticity of the *policy* and that of the *environment* [?]. A policy determines which action the agent selects in a given state [? ?]. The environment, by contrast, is modeled by transition dynamics that specify a probability distribution over the next state and reward, given the current state and action [?]. This is typically denoted as $P(s', r \mid s, a)$, where s is the current state, a is the selected action, and (s', r) is the resulting next state and reward [? ?]. However, while the decision of choosing the action of *moving right* given a state may be deterministic, what happens next is subject to the degree of stochasticity present in the environment [? ?]. Due to slipperiness, taking the action to move 'right' from that cell may result in the agent actually moving in a different direction, such as *up* or *down*, each with some probability [?]. Thus, an agent can act deterministically in an environment where the outcomes of its actions are probabilistic [? ?].

The **optimal policy** π^* is the one that maximizes the expected cumulative return from any given state [? ?]. In Markov Decision Processes (MDPs), this is often expressed as

$$\pi^*(a|s) = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \,\middle|\, \pi, s_0 = s \right] \tag{2}$$

where $\gamma \in [0,1)$ is the discount factor and r_t is the reward received at time t [? ?].

This simplicity in defining states, actions, transitions, and rewards makes toy examples like GridWorld ideal for illustrating core RL concepts, but they lack the complexity and ambiguity found in real-world applications, where the state space is vast, the actions are numerous and continuous, and the rewards are often delayed and difficult to quantify [**? ?**], .

In the context of **ad policy optimization**, the policy $\pi(a|s)$ defines how the system selects ads a to display to a user in a particular state s, using levers such as ad load, placement, and price thresholds, as detailed in Section 4.2 [**? ? ?**]. The state s may include the user's profile, browsing history, session context, and inferred commercial intent, as discussed in Section 4.3 [**? ? ?**]. The policy can be deterministic (always show the same ad for a given state) or stochastic (show ads based on probabilities) [**? ?**]. In advanced recommendation systems, ad policies are typically stochastic, whereas in older recommendation systems, policies might be deterministic (i.e. every X impressions of organic items show ad Y from list Z) [**? ?**]. **Stochastic policies** are desirable because they **naturally support exploration**, allowing the system to sample from a range of actions rather than always selecting the historically best-performing one [**? ?**]. This **helps avoid local optima** and uncover ad-user pairings that may lead to better long-term outcomes [**? ?**]. Moreover, stochastic policies provide a practical mechanism for managing uncertainty and partial information [**? ?**]. When it is unclear which action best serves all objectives, such as monetization and long-term engagement, randomizing over plausible actions allows the system to spread risk and continue learning [**? ?**]. While **deterministic policies** can in principle encode complex tradeoffs, they may hard-code assumptions that **do not generalize well or require re-tuning as user behavior shifts** [**?**]. Stochastic policies, by contrast, can naturally adapt via probabilistic weighting as more data becomes available [**? ?**].

Learning a policy involves a series of critical design decisions that must account for both **algorithmic trade-offs and real-world constraints**. These include whether to learn from historical data in an **offline** setting or through direct interaction with the environment via **online learning**; whether to rely on **value-based methods or policy-based optimization**; whether to incorporate an **explicit model of the environment or adopt a model-free approach**; and whether to use **tabular methods or generalize through function approximation**. In addition, selecting an appropriate **exploration** strategy is essential for balancing the trade-off between learning new behaviors and **exploiting** known high-reward actions. This section provides guidance on how to navigate these decisions, using the FrozenLake environment as a conceptual reference point and highlighting real-world implementations from large-scale recommendation systems.

4.4.1 Offline vs Online Policy Learning

Depending on whether the agent learns by training on historical data or by interacting with the environment in real time, **policy learning can be categorized as either offline or online [?]**. **In offline policy learning,** models are trained using logged data from past user interactions, without actively modifying the live system [**?**]. This is often the safest and most scalable approach in large platforms, as it avoids the risk of deploying untested policies [**?**]. Offline methods are particularly useful during the early stages of development when real-time exploration is expensive [**? ? ?**].

Monte Carlo (MC) methods estimate the expected return of a policy by averaging cumulative rewards across complete episodes of logged interactions [**? ?**]. Because they require observing full trajectories before updates, MC methods are particularly well-suited to settings where rewards are delayed and live exploration is costly [**? ?**]. This makes them a natural fit for early-stage offline learning scenarios, where the focus is on extracting long-term value insights from existing data without interacting with the live environment [**? ?**]. However, Monte Carlo methods have notable limitations [**?**]. Waiting for full episodes to complete can make them highly sample-inefficient, particularly in environments where episodes are long or rewards are sparse [**?**]. This inefficiency can delay policy improvements and slow adaptation to changes [**?**]. Furthermore, MC estimates often suffer from high variance, especially in noisy or stochastic environments, which can destabilize learning unless techniques like reward normalization, batch averaging, or variance reduction are applied [**?**]. Monte Carlo approaches also assume that logged episodes are representative of the environment; if the logging policy is narrow or biased, importance sampling corrections become necessary to avoid skewed estimates [**?**]. Due to these limitations, Monte Carlo methods are typically not used in high-frequency online systems where quick updates are critical [**?**]. As an illustrative example of offline learning, in the context of the FrozenLake environment, offline policy learning refers to the process of learning a policy using a log of past episodes, i.e. sequences of (state, action, next state, reward) generated by a prior policy. The learner must rely entirely on the logged data to reason about optimal behavior in navigating from the start state (S) to the goal (G), while avoiding holes (H) and dealing with slippery transitions.

Logged data only contains actions chosen by the historical policy, which means there is no direct visibility into what would have happened if alternative actions had been taken in the same context [**?**]. This **lack of**

counterfactual information makes it difficult to evaluate or iterate on new policies [?]. However, methods such as inverse propensity scoring (IPS), which re-weights logged rewards to approximate outcomes under a different policy, and doubly robust estimators, which combine IPS with model-based value predictions, can help address this challenge [? ? ?]. These approaches enable limited forms of counterfactual reasoning, especially when the new policy does not diverge too far from the one that generated the data [? ?].

A simple example of counterfactual estimation being viable in the FrozenLake environment can be illustrated as follows: suppose the agent has never made a left turn from state 4. While we may not have logged data on this specific action at this specific grid location, a function approximator—such as a neural network trained over state-action pairs—might still be able to infer the expected outcome. This is because the model could have learned to generalize from similar patterns observed in nearby states, allowing it to estimate the return of an unobserved action in a plausible way [? ?].

To overcome the constraints of limited exploration in offline settings, **online policy learning enables agents to refine their policies by interacting with the environment and learning from new experiences** [?]. The agent is able to explore different actions, observe the consequences in real time, gather feedback, and update its behavior accordingly. This ability to learn directly through interaction with the environment makes online learning more flexible and adaptive, although it also introduces greater risk [?].

To mitigate risks and costs of online exploration, offline is almost always used first [?]. Online is introduced gradually via controlled bandit loops, feature gates, or soft rollouts. This hybrid approach of offline-then-online policy learning is commonly seen in industry. In their offline RL system, YouTube trained an actor-critic model entirely on logged user interaction data, using Monte Carlo returns to estimate long-term value, and only deployed policies to production after thorough offline evaluation [?]. At Meta, the Horizon platform first trains policies offline using counterfactual estimators like Doubly Robust evaluation, and then deploys them through gated rollouts, which refer to the controlled and staged release of new policies to a limited portion of traffic in order to minimize risk during online learning [?]. Spotify similarly adopts a cautious approach by first optimizing contextual bandit policies offline, and then introducing online learning through incremental updates and careful exploration strategies [?].

4.4.2 Value-based vs Policy-based Learning

RL algorithms can be divided into value-based and policy-based methods based on how they learn to make decisions to maximize cumulative rewards [**? ?**]. Value-based methods first estimate the value of states or state-action pairs and then derive a policy from these estimates, whereas policy-based methods learn the policy directly by optimizing expected cumulative reward [**? ?**]. In other words, value-based methods focus on learning the utility of states or actions and derive the optimal behavior indirectly by acting greedily with respect to this learned value [**?**]. Policy-based methods, in contrast, directly optimize the policy without estimating value functions [**? ?**].

There are two common **value functions**. The first is the state value function, defined as

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty}\gamma^{t}R_{t+1}\,\middle|\,S_0 = s\right] \tag{3}$$

which gives the expected return when starting in state s and following policy π thereafter. As in standard RL notation, R_{t+1} is the reward received at time step $t+1$, and $\gamma \in [0,1]$ is the discount factor that determines how much future rewards are valued relative to immediate ones. The lower the value of γ, the less future rewards influence the total return [**? ?**]. The second is the action value function, defined as

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty}\gamma^{t}R_{t+1}\,\middle|\,S_0 = s, A_0 = a\right] \tag{4}$$

which gives the expected return when starting in state s, taking action a, and then following policy π [**? ?**]. Once the action-value function $Q(s,a)$ is learned, the agent can act greedily with respect to its current value estimates by selecting the action that maximizes this function:

$$\pi(s) = \arg\max_{a} Q(s,a) \tag{5}$$

where $\pi(s)$ denotes the policy derived from the current value function [**?**]. This approach allows the agent to act greedily with respect to its current value estimates, that is, to select the action that appears to yield the highest expected return at the next decision point, gradually improving its behavior as the value function becomes more accurate [?]. Because value-based methods rely on greedy action selection to determine the next best action, they require comparing the estimated value of all possible actions at each decision point (e.g. each state) [? ?]. As the number of available actions grows, and as the agent encounters more states during interaction, this exhaustive comparison becomes computationally intractable [?]. For this reason, value-based methods are particularly well-suited for environments with discrete or low-dimensional action spaces, where enumerating and comparing action values remains computationally feasible [? ?]. Common algorithms used in value-based learning include Q-learning, SARSA, and Deep Q-Networks (DQN) [? ?].

In Policy-based learning, the agent tries out different strategies and tweaks their behavior directly based on what works well overall, rather than scoring every possible move [? ?]. Formally, the goal is to learn a parameterized policy $\pi_\theta(a \mid s)$, where θ denotes the parameters of the policy [? ?]. In practice, especially in modern industrial applications, these parameters are often the weights of a neural network, though other function approximators are possible [? ? ?]. Neural network-based policies have become the norm in large-scale recommendation systems due to their expressiveness and scalability, making them particularly well-suited for environments with complex and high-dimensional state spaces [? ? ?]. Formally, the objective function to be maximized is the expected cumulative reward under the policy:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \right] \tag{6}$$

where τ denotes a trajectory: a sequence of states and actions generated by following the policy π_θ [? ?]. To optimize this objective, the policy gradient theorem provides a way to compute the gradient of the expected return with respect to the policy parameters, given by:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a \mid s) \cdot Q^{\pi_\theta}(s, a) \right] \tag{7}$$

where $Q^{\pi_\theta}(s, a)$ is the action-value function under the current policy [? ?]. Since the goal is to maximize return, gradient ascent, not descent, is used [? ?]. **Common policy-based RL algorithms** include REINFORCE, Proximal Policy Optimization (PPO), Trust Region Policy Optimization (TRPO), and various Actor-Critic methods, which **combine both policy and value learning in a hybrid architecture** [? ? ? ?].

In theory, either value-based or policy-based learning is sufficient to solve an RL problem [**? ?**]. However, policy-based methods tend to be more suitable for ad policy optimization for a number of reasons [**? ?**]. **First, policy-based methods are preferred in scenarios where stochastic policies are desired** because they explicitly parameterize the policy as a probability distribution over actions, rather than implicitly choosing a single best action [**? ?**]. What policy-based methods learn are the stochastic policy $\pi_\theta(a \mid s)$ directly, whereby the output is itself a probability distribution (e.g. softmax over action logits, Gaussian in continuous space) [**? ?**]. **Moreover, policy-based methods are also desirable when careful tuning of exploration-exploitation is required**, as they can learn when and how to be random, whereas value-based methods cannot [**? ? ?**]. In contrast, because value-based methods learn the action-value function $Q(s, a)$ and then derive the deterministic policy $\arg\max_a Q(s, a)$, value-based methods cannot directly represent probability distributions over actions [**? ?**]. To accommodate for stochasticity, value-based methods resort to ϵ-greedy approaches, which are ad hoc and not a learned methodology [**? ?**]. **Finally, Policy-based methods are often used when we want to optimize a policy across an entire ML pipeline**[**? ?**]. It is common for ad-ranking systems to start with a transformer-based encoder for ad retrieval and to conclude with a multi-head attention blending layer that predicts a downstream action, where end-to-end differentiability is required to allow the gradient to flow and therefore learn a cohesive policy [**? ? ?**]. This is difficult to achieve with value-based methods, which estimate a value function $Q(s, a)$ and then select actions via $\arg\max_a Q(s, a)$ [**? ?**]. Because $\arg\max$ is non-differentiable, it breaks gradient flow in neural networks [**? ?**]. In contrast, policy-based methods directly parameterize the policy $\pi_\theta(a \mid s)$ using differentiable functions (e.g., softmax or Gaussians) [**? ?**]. As a result, gradients can backpropagate through the entire pipeline, from the final action probabilities to earlier neural layers, enabling a fully integrated, end-to-end approach that can incorporate embeddings, attention mechanisms, and encoder-decoder architectures [**? ? ?**].

While policy-based methods are powerful and overcome many of the limitations of value-based methods, **policy-based methods also have their shortcomings** [**? ?**]. Policy-based methods assume that the performance of the gradient can be accurately estimated and optimized through sampling [**? ?**]. However, **these estimates often suffer from high variance**, which can lead to unstable learning and slow convergence [**? ?**]. Because of the absence of an explicit value function, policy-based methods tend to require a larger number of samples compared to value-based methods, making them **less sample-efficient** [**? ?**]. **To overcome these limitations, many large-scale production systems adopt hybrid approaches that combine both value-based and policy-based learning, most notably in the form of actor-critic architectures** [**? ? ?**]. In these methods, **the actor is responsible for learning the policy**, **while the critic estimates the value function to guide and stabilize the policy updates** [**? ?**]. This combination improves learning stability, speeds up convergence, and enhances interpretability, making it well-suited for complex, real-world applications such as recommendation and ad-serving systems [**? ? ?**].

To illustrate the **distinction between value-based and policy-based methods in a simple example, consider the FrozenLake environment** [**?**]. In a value-based approach, the agent learns a state-action value function $Q(s, a)$ that estimates the expected cumulative reward for each action at every grid position [**? ?**]. Once trained, the agent derives its policy by selecting the action with the highest value in each state [**?**]. For example, from a particular icy tile near a hole, it may learn that moving right has a higher expected return than moving down, based on accumulated experience. In contrast, a policy-based agent would directly learn a parameterized policy $\pi(a \mid s)$ that outputs a probability distribution over actions for each state [**? ?**]. Instead of estimating the value of each action, it adjusts its parameters to increase the likelihood of action sequences that successfully reach the goal [**? ?**]. Over time, this agent may learn, for instance, that taking a probabilistic mix of "right" and "down" from a slippery tile leads to more consistent success, even if it cannot explicitly quantify the expected value of each option.

The value-based vs policy-based taxonomy offers a useful lens for categorizing RL approaches in ad policy optimization, based on how decisions are learned to maximize cumulative rewards [**? ? ?**]. **Platforms like LinkedIn and Google Ads have applied value-based approaches like Q-learning to optimize between discrete ad policies**, such as selecting between low or high ad load treatments, by estimating the expected value of each action for a given user segment [**? ? ?**]. **Value-based learning works well when the action space is discrete, small, and easily enumerable**, allowing the agent to derive a policy through the maximization of the learned Q-function [**? ?**]. **In contrast, policy-based approaches are better suited for continuous, high-dimensional, or unstructured decision spaces** [**? ? ?**]. While industry examples above from Google and LinkedIn have applied value-based methods such as Q-learning to optimize among a small set of discrete ad policies (e.g., choosing between high and low ad load treatments) [**? ?**], it is also **possible to optimize for the ad policy directly in a more flexible formulation** [**? ? ?**]. For instance, rather than predefining a limited set of treatment arms, TikTok used Proximal Policy Optimization (PPO) to learn a stochastic ranking policy that directly maps user context to slates of ads [**? ?**]. This approach can handle potentially large or complex sets of ads more flexibly, allowing for direct parameterization of the stochastic policy [**? ? ?**]. The built-in stochasticity facilitates exploration (helping to uncover effective ad configurations) and adaptation to shifting user behaviors, without constraining the policy to a small, predefined action space [**? ? ?**].

Actor-critic architectures present a hybrid method that leverages the benefits of both value-based and policy-based methods [?]. The actor learns the policy, while the critic provides value estimates to stabilize and accelerate learning [? ?]. The critic acts as a baseline to reduce the high variance of policy gradient methods (e.g. REINFORCE), and the actor's explicit policy handles large or complex action spaces better than value-based approaches (e.g. DQN) [**? ? ?**].

There are well-documented industry examples of the use of actor-critic methods in ad recommendations. For example, Facebook reported that RL models (including off-policy actor-critic methods) trained on its Horizon platform outperformed and even replaced previous supervised recommendation systems in tasks like notifications and video ranking [**?**]. YouTube introduced an off-policy actor-critic approach to augment their previous REINFORCE agent (policy-gradient based) by adding a critic network that estimates Q-values via temporal difference (TD) learning [**?**]. The use of Q-value estimates enables lower-variance policy updates compared to Monte Carlo returns; one-step importance correction offers more robust handling of off-policy data than full-trajectory importance sampling; and bootstrapping with one-step TD targets improves the estimation of long-term rewards [**?**].

Actor-critic methods are particularly useful in multi-objective settings where both revenue and user engagement must be balanced, and where using logged data for offline training is desired [**?**]. To this end, two-staged constrained actor-critic methods have been effectively used to optimize the main objective while enforcing guardrail constraints through a constrained MDP formulation [**?**]. In stage one, multiple actor-critic agents learn to optimize auxiliary objectives (e.g. engagement such as likes, saves, and shares) [**?**]. In stage two, a primary actor-critic learns a policy maximizing the main objective (e.g. revenue, ad clicks) while staying close to the auxiliary policies to satisfy those constraints. This two-tier actor-critic approach achieved a superior balance: offline experiments showed it outperformed alternative RL methods on the trade-off between watch time and other interaction metrics [?].

Actor-critic RL brings the bias–variance trade-off under control: the critic (value-based element) introduces bias but provides low-variance evaluation, and the actor (policy-based element) ensures we still directly optimize the true objective [? ?]. Understanding the structure and cardinality of the action and state spaces, the importance of calibrating exploration vs exploitation, the available computational budget, and the degree of cohesion required across an ML pipeline are some key factors that help determine which paradigm to use in practice [? ? ?].

4.5 Balancing Exploration and Exploitation in Policy Learning

Balancing exploration and exploitation is a fundamental challenge in policy learning [?], particularly at large scale, whereby exhaustively exploring all options is computationally infeasible, the cost of suboptimal decisions compounds quickly, and sustained bias toward early winners can prevent the discovery of truly optimal policies over time [?].

In ad policy optimization, tuning exploration and exploitation is critical across multiple dimensions, notably ad selection and ad load. In *ad selection*, a system must decide between exploring new ads that might yield higher rewards and exploiting ads known to perform well [? ? ?]. In setting *ad load* configuration, platforms must weigh the exploration of higher ad exposure for queries with high commercial intent against the risk of degrading user experience [? ? ?]. While exploring more aggressive ad load configurations may unlock revenue gains, it risks triggering ad fatigue, reduced engagement, and eventual user attrition [? ?].

In this section, we introduce the commonly used methods in industry as well as their pros and cons, and the contexts in which they are most effective.

4.5.1 ϵ-Greedy Strategy

One of the simplest and most widely used exploration methods, the epsilon-greedy strategy allows the agent to primarily exploit known successful actions (with probability $1 - \epsilon$) while occasionally exploring random actions (with probability ϵ) [**? ?**]. Over time, ϵ *can* be reduced, shifting focus toward exploitation as the agent gains more experience [**? ?**]. However, deciding how much and when to decrease ϵ is non-trivial, as reducing exploration too quickly can lead to premature convergence on suboptimal actions [**? ?**].

ϵ-Greedy based exploration/exploitation is a relatively easy method to implement, computationally efficient, and thus well-suited for real-time applications [**? ?**]. However, because it treats all actions uniformly during exploration, ϵ-greedy does not differentiate between tried and untried actions, account for uncertainty in value estimates, or prioritize actions with higher potential expected improvement [**? ?**]. This can lead to missed opportunities, especially in environments with large action spaces, because as the number of possible actions increases, the probability of selecting any particular action during exploration decreases [**? ?**]. As a result, many potentially high-value actions may never be explored, or may be explored too infrequently to accurately estimate their value [**? ?**]. ϵ-greedy is most effective in smaller action spaces, where exploration remains tractable and the risk of encountering extremely low-value actions is lower due to the reduced likelihood of extreme outcomes [**? ?**].

*4.5.2 **Upper Confidence Bound (UCB)***

In contrast to ϵ-greedy strategies, which explore uniformly at random and do not incorporate knowledge of an action's estimated value or the uncertainty in that estimate, the UCB algorithm ranks actions using a composite score that integrates both expected reward and uncertainty [**?**]. At each time step t, UCB selects the action a that maximizes:

$$\mathrm{UCB}_t(a) = \hat{Q}_t(a) + \alpha \cdot \sqrt{\frac{\log t}{N_t(a)}} \tag{8}$$

where **the first term $\hat{Q}_t(a)$ is the estimated average reward of action a, and the second term is called the exploration bonus**, which reflects the uncertainty in the reward estimate [**? ?**]. $N_t(a)$ is the number of times action a has been selected, which shrinks the bonus as the action is chosen more often and reduces exploration as a result [**? ?**]. Similarly, $\log t$ grows slowly over time, which ensures continued exploration in later rounds. This formulation with the exploration and exploitation terms ensures that actions are prioritized either because they have performed well (high $\hat{Q}_t(a)$) or because they are underexplored (high uncertainty bonus) [**? ?**]. α is a tunable parameter that determines how much exploration is valued over exploitation [**? ? ?**]. As an action is selected more frequently, its estimate becomes more reliable, uncertainty decreases, and the confidence interval narrows [**? ?**]. As a result, the UCB score may increase or decrease: if the action is good, its estimated reward improves and the score may remain high or even rise despite the shrinking bonus; if the action is bad, the estimate stays low and the diminishing uncertainty term no longer inflates its score, causing it to be deprioritized over time [**? ?**].

The unified scoring function in UCB naturally balances exploration and exploitation [? ?]. It allows the agent to identify promising but underexplored actions while avoiding those that are confidently known to perform poorly [**?**]. While UCB provides tighter bounds on long-term regret, it can be more computationally expensive than simpler methods like ϵ-greedy and may over-explore suboptimal actions with high uncertainty, leading to short-term performance losses [**? ? ?**].

LinkedIn used UCB-style methods to optimize ad placement and pacing in user feeds. In the early stages of their system, UCB was employed to evaluate which treatment arms, such as different spacing, reserve price, and ad load configurations, performed best across user segments. UCB helped balance exploration of new pacing parameters with exploitation of those already showing promising results [**?**]. **Microsoft, through the Contextual Bandit Bake-Off, found that UCB and its linear variant, LinUCB, were competitive or superior to ϵ-greedy approaches across multiple offline datasets for ad personalization and content recommendation** [**?**]. **Facebook (Meta) incorporated UCB-inspired bandit methods within its Horizon RL platform, particularly for notification ranking and ad sequencing tasks**. These methods were used in early-stage exploration, particularly in cold-start scenarios or on surfaces with limited historical data, before transitioning to more complex actor-critic architectures for long-term optimization [**?**].

UCB performs best in environments where data is sparse, **optimization horizons are long**, and **the cost of regret is high**, making it well-suited for ad policy optimization where long-term performance (like maximizing user lifetime value or engagement) is key [**? ? ?**]. However, in real-time applications where immediate results are prioritized, UCB's tendency to over-explore may be less suitable [**? ? ?**]. Additionally, the choice of the exploration parameter α can be brittle as too conservative a setting may limit the discovery of better actions, while too aggressive a setting may amplify noise and degrade performance.

*4.5.3 **Thompson Sampling***

Thompson Sampling (TS) uses a Bayesian approach in which actions are selected by sampling from the posterior distribution over rewards and choosing the action with the highest sampled value [**?** **?**]. Formally, the selected action at time t is:

$$a_t = \arg\max_{a \in \mathcal{A}} \tilde{Q}_t(a) \tag{9}$$

Formally, the selected action at time t is: where $\mathcal{A}$ is the action set and $\tilde{Q}_t(a)$ is a sample from the posterior distribution of the expected reward for action a at time t [**?**].

While each individual decision is based on a single random sample, across many realizations, the likelihood that an action is selected reflects the posterior probability that it is the optimal choice [**?**]. Initially, TS relies on a prior distribution that reflects the initial belief about the reward of each action, usually based on some default assumptions or domain knowledge [**?**]. As more data is collected through interactions (e.g., clicks or conversions in an ad setting), the prior is updated to form a posterior distribution, representing the system's refined belief about expected rewards [**?**]. Actions with greater uncertainty (i.e., wider posterior distributions) are more likely to generate extreme reward samples, occasionally leading to their selection even if their mean reward is lower [**?**]. In contrast, actions with higher expected rewards and narrower posterior distributions are selected more consistently due to their reliably higher sampled values [**?**]. Thus, TS leverages posterior variance to simultaneously enable exploration and exploitation without requiring an explicit exploration bonus as used in UCB methods [**?**].

While both UCB and TS leverage estimates of variability to manage exploration and exploitation, they differ in how they quantify uncertainty, handle randomness, and select actions [**?** **?**]. UCB constructs an upper confidence bound for each action and always selects the action with the highest upper bound [**?**], whereas TS samples from the posterior distribution over rewards and chooses the action with the highest sampled value [**?**]. As a result, UCB deterministically selects the same action given the same inputs, while TS's choice varies across runs due to its sampling-based nature [**?**]. Although UCB is often easier to implement, as it does not require specifying prior or posterior distributions, it can be more brittle to tune (e.g., selecting an appropriate exploration constant α) [**? ?**]. In contrast, TS is generally more robust to hyperparameter choices and tends to perform better in non-stationary environments [**? ?**].

Studies by Chapelle and Li [**?**] show that TS can perform as well as, or better than, UCB in terms of cumulative regret, defined as the total loss incurred from not consistently selecting the optimal action [**? ?**]. This is especially true in **scenarios where data is sparse or feedback is delayed [?]**. In contrast to UCB, which might over-explore uncertain actions with low potential returns, TS ensures a more balanced and probabilistic form of exploration [**?**]. TS is naturally well-suited for industry problems like selecting ads for new or existing users, particularly in dynamic environments such as online advertising, where user preferences shift frequently and new content is introduced continuously. However, because TS relies on randomized sampling, it can exhibit high variance, which may lead to less stable performance in environments that prioritize short-term accuracy or require highly reliable decision-making [**?**].

*4.5.4 **Broader Considerations for Exploration and Exploitation***

While the previous discussion focused on canonical exploration strategies that are widely deployed and highly effective in industrial recommender and ad systems (e.g., ϵ-Greedy, UCB, Thompson Sampling), it is also important to recognize that additional techniques are often employed to address more complex, high-dimensional, or rapidly evolving environments. For instance, in dynamic ad allocation systems, parameters are often tuned in real-time to balance short-term revenue generation against user experience metrics such as Click Yield, which measures the rate of user interactions per ad impression [**?**]. By adjusting trade-off parameters in response to shifting engagement patterns, platforms can dynamically prioritize either immediate monetization or sustained user satisfaction. Moreover, alternative approaches such as entropy-regularized policies [**?**], Bayesian optimization [**?**], and policy gradient methods like Proximal Policy Optimization (PPO) [**?**] extend the exploration-exploitation framework in settings where continuous adaptation is critical.

5 Conclusions

The integration of RL in policy optimization for large-scale recommender systems has gained popularity as a promising approach to balance long-term and short-term objectives in environments with complex user behavior and revenue dynamics. However, scaling RL in practice presents challenges, including vast state and action spaces, reliance on proxy metrics, and significant computational demands. This paper provides an overview of effective, scalable techniques that address these challenges, offering insights into how RL can be applied at scale. Unlike traditional methods that focus on optimizing short-term outcomes, RL's cumulative reward framework enables it to account for both immediate and delayed feedback. Together, contextual bandits and offline replay offer scalable, interpretable solutions that approximate full RL while managing real-time constraints. Contextual bandits support selecting optimal actions within each interaction based on the current state, maximizing immediate rewards, which are typically proxy metrics of the true long-term reward. Offline replay complements this by enabling counterfactual learning: multiple treatments can be tested across randomized populations to evaluate causal effects, refining policies for diverse user segments.

Future research may focus on enhancing interpretability, refining reward functions, and exploring decentralized approaches to handle complex decision layers. These advancements will bring RL closer to fulfilling its potential in providing tailored, dynamic ad recommendations that align user satisfaction with business goals in large-scale recommendation systems.

Acknowledgements

I would like to thank my advisor, Yuri Lawryshyn, for his valuable guidance and direction throughout this work. I also extend my gratitude to Adam Obeng, Bee-Chung Chen, and Jay Adams for their critical coaching and insights, which have significantly shaped my approach and understanding of industry practices. Their expertise has been key in aligning academic research with practical applications.

Component of Markov Decision Process Formulation	Frozen Lake Example	Large Scale Recommendation Systems for Ad Policy Optimization
Environment	The frozen lake.	The users.
Goal	Reach the goal (G) from the start position (S) while avoiding holes (H). The agent must avoid falling into a hole, and reaching the goal faster is rewarded.	Maximize revenue without compromising user experience.
Episode	A game.	A time period (T), typically 2 weeks, corresponding to an A/B experiment where enough data is collected to evaluate policy performance across users.
Rewards	Rewarded for reaching the goal (i.e., +100), but penalized for delays (i.e., -1 per step).	The reward is typically represented by a utility function, which could be a weighted sum of metrics like user engagement (clicks, time spent) and revenue generated (ad clicks, conversions). The challenge lies in balancing short-term revenue with long-term user retention.
Agent	The learner.	The Recommendation System.
Actions	Up, down, left and right.	Basically infinite. But when it comes to ad recommendations specifically, some typical actions are the following: • Reserve Price Floor: The minimum price at which an ad impression is sold. • Displacement Cost Threshold: A measure of the opportunity cost for showing an ad. • Max Ad Load: The maximum number of ads that can be shown without reducing user satisfaction. • Ad Quality Thresholds: Standards used to ensure ads meet a quality bar before being shown. • Ad Placement: The specific position of ads within the user feed (e.g., first, third, or fifth position).
States	All possible grid positions the agent can occupy, representing the state space of the environment.	**Signals**: • User Context: Demographics, user profile • Interaction Data: Previous interactions with ads, organic content • Contextual Features: Time of day, browsing context • Embeddings: Latent feature representations of user interests or ad features
Policy	A mapping of grid positions to optimal action to take for a maximum reward.	A mapping of user segments to ad treatments to maximize both revenue and user experience. This policy can be dynamically adjusted based on real-time feedback or offline simulations.

Table 1: Comparison of Frozen Lake Example and Large-Scale Recommendation Systems for Ad Policy Optimization

Objective	**Proxy Metric**	**Short Description**
Revenue	**CTR (Click-Through Rate)**	Approximates revenue potential; higher CTR often correlates with higher ad clicks [135, 140, 144].
	Ad Impressions	Number of times an ad is displayed to users.
	pCTR (Predicted CTR)	Model-estimated probability that a user will click on an ad.
	Predicted Revenue	Often a function of bids and pCTR to estimate expected revenue [135, 140].
	Click Yield	Ratio of total clicks (ads + organic) to total impressions on a page. Helps avoid bias of optimizing CTR alone [140].
Engagement	**Session Duration**	Tracks how long users stay engaged per session (proxy for longer-term engagement).
	30-day Total Sessions	Aggregated count of user sessions or interactions over a 30-day window, proxy for habit or stickiness [134].
	Likes / Adds to Library	High-intent actions indicating deeper user satisfaction and potential long-term retention [80].
	Dwell Time	Actual time spent viewing/reading an item, capturing depth of engagement [138].
	Shares	Another high-intent action reflecting strong user engagement.
	Ratio-based Diversity	Proportion of unique topic clusters in consumption history (indicates breadth of user interest) [126].
	High-quality Consumption	Tracks consumption of content with high engagement (e.g., full video views, long reads) [126].
	Repeated Consumption	Monitors how often users revisit similar content, reflecting satisfaction [126].
	Time to Revisit	Interval between consecutive visits; shorter intervals suggest higher retention [126].
	Distribution-based Diversity	Uses entropy to measure spread of topics consumed (gauges exploration vs. narrow focus) [126].
	User Return Rate (URR)	Fraction of users who revisit the platform after a set interval, reflecting long-term retention [130]
	Relative Watch Time	Percentage of a video watched, normalized by video length (better gauge of true engagement than raw views) [131].
Ad Fatigue	**Depth**	Reduction in engagement (e.g., fewer impressions per session) over time [100].
	Length	Decrease in time spent per session [100].
	Frequency	Decrease in number of sessions per week [100].
	Ad Load (Sessions)	Number of sessions that end (or quit) during an ad, indicating ad overload [100].
	Ad Load (Absolute)	Absolute ad volume across sessions and changes in that volume [100].
	Click Entropy	Measures uncertainty/diversity in ad clicks for a query; high entropy may imply users find ads less relevant [100].

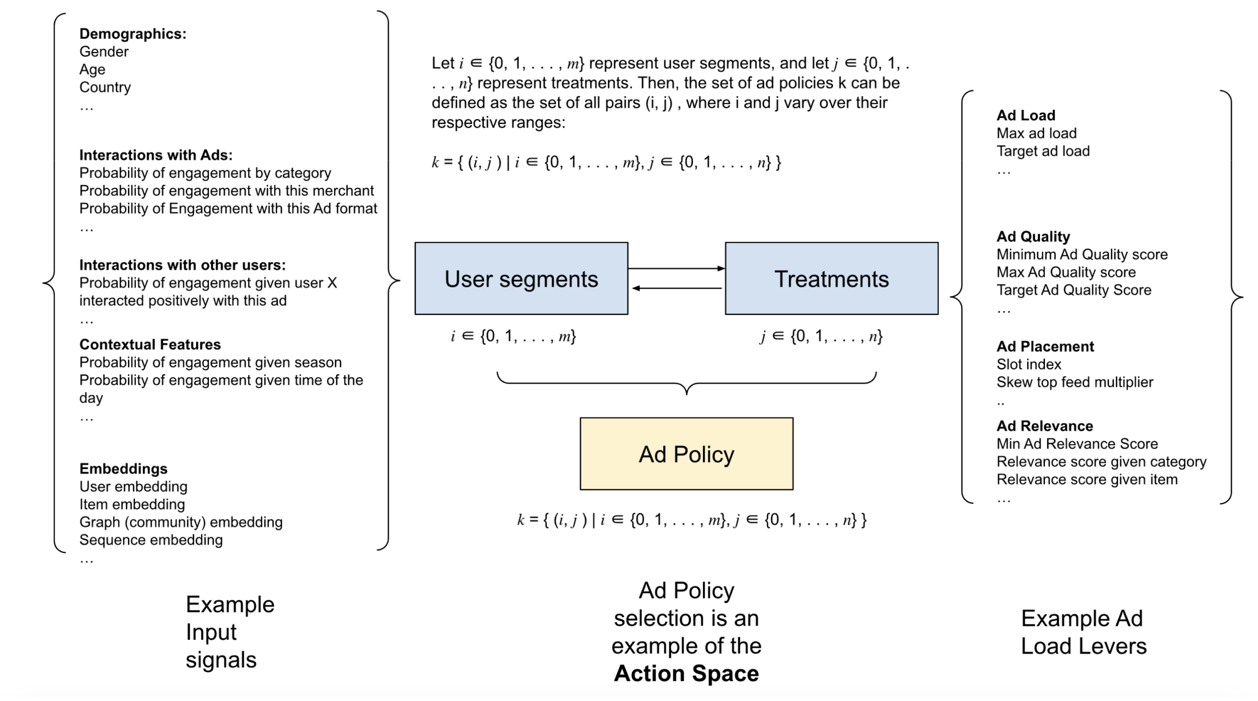
Figure 3: Proxy metrics for Revenue, Engagement, and Ad Fatigue.

Company	Paper	Organic Content Reward/Utility	Ads Reward/Utility	Blended Reward/Utility
TikTok	[140]	User Experience (r_t^{ex}): Measures the negative impact of displaying an ad on the user's experience, with a binary outcome. A positive reward (1) is given if the user continues browsing after seeing the ad, while a negative reward (-1) is given if the user leaves the platform after seeing the ad.	Ad Income (r_t^{ad}): Represents the immediate revenue generated from displaying an ad. It provides a positive value if an ad is displayed, contributing to maximizing advertising income.	$r_t(s_t, a_t) = r_t^{ad} + \alpha \cdot r_t^{ex}$ Here, α is a hyperparameter that controls the trade-off between maximizing ad revenue and minimizing negative user experience.
TikTok	[142]	$R_{\text{rec}}(s_t, a_t)$ Engagement metric (e.g., dwell time, clicks, purchases)	$R_{\text{ad}}(s_t, a_t) = \begin{cases} 1 & \text{if user stays after ad} \\ 0 & \text{if user leaves after ad} \end{cases}$	$R_{\text{total}}(s_t, a_t) = \alpha \cdot R_{\text{rec}}(s_t, a_t) + \beta \cdot R_{\text{ad}}(s_t, a_t)$
Alibaba	[138]	Click Yield (CY): Measures the overall performance of the entire page, considering both ads and organic content. It is defined as the ratio of total clicks on all items (both ads and organic) over the total number of impressions. $$CY_{ij} = \frac{\sum_{n=1}^{N} Click(i, j, n)}{Imp(i, j)}$$ Where $Click(i, j, n)$ is the click for item n in list j, and $Imp(i, j)$ is the number of impressions.	Revenue (REV): For sponsored content, revenue is computed based on the click-through rate (CTR) and cost per click (CPC). $$REV_{ij} = \sum_{n=1}^{N} CTR(i, j, n) \times CPC(i, j, n)$$ The CPC is determined by the auction mechanism (usually Generalized Second Price Auction).	The system optimizes the trade-off between revenue (REV) and Click Yield (CY). The optimization problem is defined as: $$\max \sum_{i,j} REV_{ij} \cdot x_{ij}$$ Subject to the constraint: $$\frac{\sum_{i,j} CY_{ij} \cdot x_{ij}}{\sum_{i,j} x_{ij}} \geq T$$ Where T is a threshold for the average Click Yield, ensuring user satisfaction.
LinkedIn	[134]	Expected Engagement Utility: Measures user actions like clicks, comments, shares, or time spent on organic content. The goal is to maximize engagement with organic content.	Expected Revenue Utility (for ads): Evaluates the revenue generated from ads, typically calculated as the product of the bid amount and the predicted click-through rate (pCTR).	The utility function for ads and organic content is formulated as: $$\max \sum_i x_i r_i - \frac{w}{2} \lVert x \rVert^2$$ Subject to the engagement constraint: $$\sum_i x_i u_{ai} + (1 - x_i) u_{oi} \geq C$$ Where: • x_i is a decision variable determining whether to place an ad in position i, • r_i represents the revenue utility for ad i, • u_{ai} and u_{oi} represent the engagement utilities for ads and organic content, respectively, • C is the engagement threshold. The shadow bid α acts as a conversion factor to blend the two objectives by equating engagement utility and revenue in a comparable manner.

Figure 4: Comparison of Utility/Reward Formulation Approaches across Different Companies

Method	How It Works	Pros	Cons	Best Use Cases
Epsilon-Greedy	Selects the best-known action with probability $1 - \epsilon$ and explores randomly with probability ϵ.	• Simple to implement • Low computational cost • Easy to understand	• Fixed exploration rate • Might miss better actions • Not adaptive to context	• Simple, static environments • Prioritize computational efficiency
Upper Confidence Bound (UCB)	Selects actions based on confidence intervals combining reward estimates and uncertainty.	• Systematic exploration • Minimizes long-term regret • Strong theoretical foundations	• May overexplore uncertain actions • Higher computational complexity	• Real-time ad placement • Long-term optimization
Regret Confidence Bounds (RegCB)	Extends UCB by using regression oracles to compute confidence intervals based on contextual features.	• Handles high-dimensional, sparse data • More accurate reward estimates • Well-suited for complex environments	• High computational cost • Difficult to scale	• High-dimensional environments • Needs precise reward prediction
Thompson Sampling (TS)	Uses Bayesian sampling from posterior distributions to estimate optimal actions.	• Adapts quickly to new data • Naturally balances exploration • Effective for cold-start problems	• Can exhibit high variance • Moderate computational complexity	• Cold-start and sparse data • Dynamic, shifting environments

Table 2: Comparison of Exploration-Exploitation Methods in Ad Policy Optimization


Let $i \in \{0, 1, \ldots, m\}$ represent user segments, and let $j \in \{0, 1, \ldots, n\}$ represent treatments. Then, the set of ad policies k can be defined as the set of all pairs (i, j) , where i and j vary over their respective ranges:

$k = \{ (i, j) \mid i \in \{0, 1, \ldots, m\}, j \in \{0, 1, \ldots, n\} \}$

Demographics:
Gender
Age
Country
…
Interactions with Ads:
Probability of engagement by category
Probability of engagement with this merchant
Probability of Engagement with this Ad format
…
Interactions with other users:
Probability of engagement given user X interacted positively with this ad
…
Contextual Features
Probability of engagement given season
Probability of engagement given time of the day
…
Embeddings
User embedding
Item embedding
Graph (community) embedding
Sequence embedding
…
User segments
Treatments
$i \in \{0, 1, \ldots, m\}$
$j \in \{0, 1, \ldots, n\}$
Ad Policy
$k = \{ (i, j) \mid i \in \{0, 1, \ldots, m\}, j \in \{0, 1, \ldots, n\} \}$
Ad Load
Max ad load
Target ad load
…
Ad Quality
Minimum Ad Quality score
Max Ad Quality score
Target Ad Quality Score
…
Ad Placement
Slot index
Skew top feed multiplier
..
Ad Relevance
Min Ad Relevance Score
Relevance score given category
Relevance score given item
…
Example Input signals
Ad Policy selection is an example of the **Action Space**
Example Ad Load Levers

Figure 5: Example of an action set in the context of ad policies in recommendation systems