Contents

1 Chapter 1 — Search Ranking as Optimization: From Business Goals to RL

Vlad Prytula

1.1 1.1 The Problem: Balancing Multiple Objectives in Search

A concrete dilemma. A pet supplies retailer faces a challenge. User A searches for "cat food"—a price-sensitive buyer who abandons carts if shipping costs are high. User B issues the same query—a premium shopper loyal to specific brands, willing to pay more for quality. The current search system shows them **identical rankings** because boost weights are static, tuned once for the "average" user. User A sees expensive premium products and abandons. User B sees discount items and questions the retailer's quality. Both users are poorly served by a one-size-fits-all approach.

The business tension. Every e-commerce search system must balance competing objectives:

- **Revenue (GMV)**: Show products users will buy, at good prices
- **Profitability (CM2)**: Prioritize items with healthy margins
- **Strategic goals (STRAT)**: Expose new products, house brands, or inventory to clear
- **User experience**: Maintain relevance, diversity, and satisfaction

Traditional search systems rely on **manually tuned boost parameters**: category multipliers, price/discount bonuses, profit margins, strategic product flags. A typical scoring function looks like:

$$s(p, q, u) = r_{\text{ES}}(q, p) + \sum_{k=1}^{K} w_k \phi_k(p, u, q) \tag{1.1}$$

{#EQ-1.1}

where $r_{\text{ES}} : \mathcal{Q} \times \mathcal{P} \to \mathbb{R}_+$ is a **base relevance score** function (e.g., BM25 from Elasticsearch or neural embeddings) mapping query-product pairs to non-negative reals, $\phi_k : \mathcal{P} \times \mathcal{U} \times \mathcal{Q} \to \mathbb{R}$ are **engineered features** (margin, discount, bestseller status, category match), and $w_k \in \mathbb{R}$ are **manually tuned weights**. Note that we've made the user dependence explicit: $s(p, q, u)$ depends on product p, query q, **and user** u through the feature functions ϕ_k.

Why manual tuning fails. The core problem: w_k cannot adapt to context. The "price hunter" (User A) cares about bulk pricing and discounts. The "premium shopper" (User B) values quality over price. A generic query ("cat food") tolerates exploration; a specific query ("Royal Canin Veterinary Diet Renal Support") demands precision.

Numerical evidence of the problem. Suppose we tune $w_{\text{discount}} = 2.0$ to maximize average GMV across all users. For price hunters, this works well—they click frequently on discounted items. But for premium shoppers, this destroys relevance—they see cheap products ranked above their preferred brands, leading to zero purchases and session abandonment. Conversely, if we tune $w_{\text{discount}} = 0.3$ for premium shoppers, price hunters see full-price items and also abandon.

Manual weights are **static, context-free, and suboptimal** by design. We need weights that adapt.

Our thesis: Treat $\mathbf{w} = (w_1, \ldots, w_K) \in \mathbb{R}^K$ as **actions to be learned**, adapting to user and query context via reinforcement learning.

In Chapter 0 (Motivation: Your First RL Experiment), we built a tiny, code-first prototype of this idea: three synthetic user types, a small action grid of boost templates, and a tabular Q-learning agent that learned context-adaptive boosts. In this chapter, we strip away implementation details and **formalize and generalize** that experiment as a contextual bandit with constraints.

> **Notation**
>
> Throughout this chapter: - **Spaces**: $\mathcal{X}$ (contexts), $\mathcal{A}$ (actions), Ω (outcomes), $\mathcal{Q}$ (queries), $\mathcal{P}$ (products), $\mathcal{U}$ (users) - **Distributions**: ρ (context distribution over $\mathcal{X}$), $P(\omega \mid x, a)$ (outcome distribution) - **Probability**: $\mathbb{P}$ (probability measure), $\mathbb{E}$ (expectation) - **Real/natural numbers**: $\mathbb{R}$, $\mathbb{N}$, $\mathbb{R}_+$ (non-negative reals) - **Norms**: $\|\cdot\|_2$ (Euclidean), $\|\cdot\|_\infty$ (supremum) - **Operators**: $\mathcal{T}$ (Bellman operator, introduced in Chapter 3)
>
> We index equations as EQ-X.Y, theorems as THM-X.Y, definitions as DEF-X.Y, remarks as REM-X.Y, and assumptions as ASM-X.Y for cross-reference. Anchors like `{#THM-1.7.2}` enable internal linking.

This chapter establishes the mathematical foundation: we formulate search ranking as a **constrained optimization problem**, then show why it requires **contextual decision-making** (bandits), and finally preview the RL framework we'll develop.

1.2 1.2 From Clicks to Outcomes: The Reward Function

Let's make the business objectives precise. Consider a single search session:

1. **User** u with segment $\sigma \in \{\text{price\_hunter}, \text{pl\_lover}, \text{premium}, \text{litter\_heavy}\}$ issues **query** q
2. System scores products $\{p_1, \ldots, p_M\}$ using boost weights $\mathbf{w}$, producing ranking π
3. User examines results with **position bias** (top slots get more attention), clicks on subset $C \subseteq \{1, \ldots, M\}$, purchases subset $B \subseteq C$
4. Session generates **outcomes**: GMV, CM2 (contribution margin 2), clicks, strategic exposures

We aggregate these into a **scalar reward**:

$$R(\mathbf{w}, u, q, \omega) = \alpha \cdot \text{GMV}(\mathbf{w}, u, q, \omega) + \beta \cdot \text{CM2}(\mathbf{w}, u, q, \omega) + \gamma \cdot \text{STRAT}(\mathbf{w}, u, q, \omega) + \delta \cdot \text{CLICKS}(\mathbf{w}, u, q, \omega) \tag{1.2}$$

{#EQ-1.2}

where ω represents the stochastic user behavior conditioned on the ranking $\pi_{\mathbf{w}}(u, q)$ induced by boost weights $\mathbf{w}$, and $(\alpha, \beta, \gamma, \delta)$ are **business weight parameters** reflecting strategic priorities. The outcome components (GMV, CM2, STRAT, CLICKS) depend on the full context $(\mathbf{w}, u, q)$ through the ranking, though we often abbreviate this dependence when clear from context.

Remark (connection to Chapter 0). The Chapter 0 toy used a simplified instance of this reward with $(\alpha, \beta, \gamma, \delta) \approx (0.6, 0.3, 0, 0.1)$ and no explicit STRAT term. All analysis in this chapter applies to that setting.

Key insight: R depends on $\mathbf{w}$ **indirectly** through the ranking π induced by scores from #EQ-1.1. A product ranked higher gets more exposure, more clicks, and influences downstream purchases. This is **not a simple function**—it's stochastic, nonlinear, and noisy.

1.2.1 Constraints: Not All Rewards Are Acceptable

High GMV alone is insufficient. A retailer must enforce **guardrails**:

$$\mathbb{E}[\text{CM2} \mid \mathbf{w}] \geq \tau_{\text{margin}} \tag{1.3a}$$

{#EQ-1.3a}

$$\mathbb{E}[\text{Exposure}_{\text{strategic}} \mid \mathbf{w}] \geq \tau_{\text{strat}} \tag{1.3b}$$

{#EQ-1.3b}

$$\mathbb{E}[\Delta\text{rank@}k \mid \mathbf{w}] \leq \tau_{\text{stability}} \tag{1.3c}$$

{#EQ-1.3c} {#EQ-1.3}

where the notation $\mathbb{E}[\cdot \mid \mathbf{w}]$ denotes expectation over stochastic user behavior ω and context distribution $\rho(x)$ when action (boost weights) $\mathbf{w}$ is applied, i.e., $\mathbb{E}[\text{CM2} \mid \mathbf{w}] := \mathbb{E}_{x\sim\rho,\omega\sim P(\cdot|x,\mathbf{w})}[\text{CM2}(\mathbf{w}, x, \omega)]$.

- **CM2 floor** (1.3a): Prevent sacrificing profitability for revenue
- **Exposure floor** (1.3b): Ensure strategic products (new launches, house brands) get visibility
- **Rank stability** (1.3c): Limit reordering volatility (users expect consistency)

This is a **constrained Markov decision process (CMDP)**, though we'll start with the simpler **contextual bandit** (single-step) formulation.

Now we understand the complete optimization problem: maximize the scalar reward #EQ-1.2 subject to constraints #EQ-1.3. This establishes what we're optimizing. Next, we'll dive deep into one critical component—the engagement term—before implementing the reward function.

Code $\leftrightarrow$ Config (constraints)

Constraint-related knobs (`MOD-zoosim.config`) live in configuration so experiments remain reproducible and auditable. These implement #EQ-1.3 constraint definitions:

- Rank stability penalty weight: `lambda_rank` in `zoosim/core/config.py`
- Profitability floor (CM2): `cm2_floor` in `zoosim/core/config.py`
- Exposure floors (strategic products): `exposure_floors` in `zoosim/core/config.py`

These are surfaced in `ActionConfig` and wired into downstream modules as they mature.

1.2.2 1.2.1 The Role of Engagement in Reward Design

In practice, search objectives are **hierarchically structured**, not flat:

1. **Viability constraints** (must satisfy or system is unusable): CTR > 0, latency < 500ms, uptime $> 99.9\%$
2. **Business outcomes** (what we optimize): GMV, profitability (CM2), strategic positioning
3. **Strategic nudges** (tiebreakers for long-term value): exploration, new product exposure, brand building

Engagement (clicks, dwell time, add-to-cart actions) **straddles this hierarchy**: it is partly viability (zero clicks ⇒ dead search, users abandon platform), partly outcome (clicks signal incomplete attribution—mobile browse, desktop purchase), and partly strategic (exploration value—today's clicks reveal preferences for tomorrow's sessions).

Why include $\delta \cdot$ CLICKS in the reward?

We include $\delta \cdot \text{CLICKS}$ as a **soft viability term** in the reward function #EQ-1.2. This serves three purposes:

1. **Incomplete attribution**: E-commerce has imperfect conversion tracking. A user clicks product p on mobile, adds to cart, completes purchase on desktop 3 days later. We observe the click, but GMV attribution goes to a different session (or is lost entirely in cross-device gaps). The click is a **leading indicator** of future GMV not captured in ω.
2. **Exploration value**: Clicks reveal user preferences even without immediate purchase. If user u clicks on premium brand products but doesn't convert, we learn u is exploring that segment—valuable for future sessions. This is **information acquisition**: clicks are samples from the user's latent utility function.
3. **Platform health**: A search system with high GMV but near-zero CTR is **brittle**—one price shock or inventory gap causes catastrophic user abandonment. Engagement is a **leading indicator of retention**: users who click regularly have higher lifetime value (LTV) than those who occasionally convert high-value purchases but otherwise ignore search results.

The clickbait risk. However, δ **must be carefully bounded**. If δ/α is too large, the agent learns **"clickbait" strategies**: optimize CTR at the expense of conversion rate (CVR = purchases/clicks). The pathological case: show irrelevant but visually attractive products (e.g., cute cat toys for dog owners), achieve high clicks but zero sales, and still get rewarded due to $\delta \cdot \text{CLICKS} \gg 0$.

Practical guideline: Set $\delta/\alpha \in [0.01, 0.10]$—engagement is a *tiebreaker*, not the primary objective. We want clicks to be a **soft regularizer** that prevents GMV-maximizing policies from collapsing engagement, not a dominant term that drives the optimization.

Diagnostic metric: Monitor **conversion quality**

$$\text{CVR}_t = \frac{\sum_{i=1}^{t} \text{GMV}_i}{\sum_{i=1}^{t} \text{CLICKS}_i}$$

(cumulative GMV per click up to episode t). If CVR_t drops $> 10\%$ below baseline while CTR rises during training, the agent is learning clickbait—reduce δ immediately.

Control-theoretic analogy: This is similar to LQR with **state and control penalties**: $c(x, u) = x^\top Q x + u^\top R u$. We penalize both deviation from target state (GMV, CM2) and control effort (engagement as "cost" of achieving GMV). The relative weights Q, R encode the tradeoff. In our case, $\alpha, \beta, \gamma, \delta$ play the role of Q, and we're learning the optimal policy $\pi^*(x)$ under this cost structure. See Section 1.10 for deeper connections to classical control.

Multi-episode perspective (Chapter 11 preview): In a **Markov Decision Process (MDP)** with inter-session dynamics, engagement enters *implicitly* through its effect on retention and lifetime value:

$$V^{\pi}(s_0) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t \text{GMV}_t \mid s_0\right] \tag{1.2'}$$

{#EQ-1.2-prime}

If today's clicks increase the probability that user u returns tomorrow (state transition $s_{t+1} = f(s_t, \text{clicks}_t, ...)$), then maximizing #EQ-1.2-prime automatically incentivizes engagement. We wouldn't need $\delta \cdot \text{CLICKS}$ in the single-step reward—it would be *derived* from optimal long-term value.

However, the **single-step contextual bandit** (our MVP formulation) cannot model inter-session dynamics. Each search is treated as independent: user arrives, we rank, user interacts, episode terminates. No s_{t+1}, no retention modeling. Including $\delta \cdot \text{CLICKS}$ is a **heuristic proxy** for the missing LTV component—mathematically imperfect, but empirically essential for search systems.

The honest assessment: This is a **theory-practice tradeoff**. The "correct" formulation is #EQ-1.2-prime (multi-episode MDP), but it requires modeling complex user dynamics (churn, seasonality, cross-session preferences) that are expensive to simulate and hard to learn from. The single-step approximation #EQ-1.2 with $\delta \cdot \text{CLICKS}$ is **pragmatic**: it captures 80% of the value with 20% of the complexity. For the MVP, this is the right tradeoff. Chapter 11 extends to multi-episode settings where engagement is properly modeled as state dynamics.

Cross-reference — Chapter 11

The full multi-episode treatment and implementation live in `Chapter 11 - Multi-Episode Inter-Session MDP` (see `docs/book/drafts/syllabus.md`). There we add `zoosim/multi_episode/session_env.py` and `zoosim/multi_episode/retention.py` to operationalize #EQ-1.2-prime with a retention/hazard state and validate that engagement raises long-term value without needing an explicit $\delta \cdot \text{CLICKS}$ term.

Code $\leftrightarrow$ Config (reward weights)

Business weights in `RewardConfig` (`MOD-zoosim.config`) implement #EQ-1.2 parameters and must satisfy engagement bounds from this section:

- α (GMV): Primary objective, normalized to 1.0 by convention
- β/α (CM2 weight): Profit sensitivity, typically $\in [0.3, 0.8]$ (higher $\Rightarrow$ prioritize margin over revenue)
- γ/α (STRAT weight): Strategic priority, typically $\in [0.1, 0.3]$ (house brands, new products, clearance)
- δ/α **(CLICKS weight): Bounded** $\in [0.01, 0.10]$ **to prevent clickbait strategies**

Validation (enforced in code): see `zoosim/dynamics/reward.py` for an assertion on δ/α in the production reward path. Chapter 8 derives principled bounds from Lagrangian constraint analysis.

Diagnostic: Compute $\text{CVR}_t = \sum \text{GMV}_i / \sum \text{CLICKS}_i$ after each policy update. If CVR drops $> 10\%$ while CTR rises, reduce δ by 30–50%.

Code $\leftrightarrow$ Simulator Layout

- `zoosim/core/config.py` (`MOD-zoosim.config`): SimulatorConfig/RewardConfig with seeds, guardrails, and reward weights
- `zoosim/world/{catalog,users,queries}.py`: deterministic catalog + segment + query generation (Chapter 4)
- `zoosim/ranking/{relevance,features}.py`: base relevance and boost feature engineering (Chapter 5)
- `zoosim/dynamics/{behavior,reward}.py` (`MOD-zoosim.behavior`, `MOD-zoosim.reward`): click/abandonment dynamics + reward aggregation for #EQ-1.2
- `zoosim/envs/{search_env.py,gym_env.py}` (`MOD-zoosim.env`): single-step environment and Gym wrapper wiring the simulator together
- `zoosim/multi_episode/{session_env.py,retention.py}`: Chapter 11's retention-aware MDP implementing #EQ-1.2-prime

1.2.3 Let's Verify the Reward Function with Code

Before diving into theory, let's implement #EQ-1.2 and see what it does. We'll use a minimal simulator to generate outcomes.

```python
from dataclasses import dataclass
from typing import NamedTuple
import numpy as np

class SessionOutcome(NamedTuple):
    """Outcomes from a single search session.

    Mathematical correspondence: realization omega in Omega of random variables
    (GMV, CM2, STRAT, CLICKS).
    """
    gmv: float          # Gross merchandise value (EUR)
    cm2: float          # Contribution margin 2 (EUR)
    strat_exposure: int # Number of strategic products in top-10
    clicks: int         # Total clicks

@dataclass
class BusinessWeights:
    """Business priority coefficients (alpha, beta, gamma, delta) in #EQ-1.2."""
    alpha_gmv: float = 1.0
    beta_cm2: float = 0.5
    gamma_strat: float = 0.2
    delta_clicks: float = 0.1

def compute_reward(outcome: SessionOutcome, weights: BusinessWeights) -> float:
    """Implements #EQ-1.2: R = alpha*GMV + beta*CM2 + gamma*STRAT + delta*CLICKS.

    This is the **scalar objective** we will maximize via RL.
```

```
    See `zoosim/dynamics/reward.py:1` for the production implementation that
    aggregates GMV/CM2/strategic exposure/clicks using `RewardConfig`
    parameters defined in `zoosim/core/config.py:193`.
    """
    return (weights.alpha_gmv * outcome.gmv +
            weights.beta_cm2 * outcome.cm2 +
            weights.gamma_strat * outcome.strat_exposure +
            weights.delta_clicks * outcome.clicks)

# Example: Compare two strategies
# Strategy A: Maximize GMV (show expensive products)
outcome_A = SessionOutcome(gmv=120.0, cm2=15.0, strat_exposure=1, clicks=3)

# Strategy B: Balance GMV and CM2 (show profitable products)
outcome_B = SessionOutcome(gmv=100.0, cm2=35.0, strat_exposure=3, clicks=4)

weights = BusinessWeights(alpha_gmv=1.0, beta_cm2=0.5, gamma_strat=0.2, delta_clicks=0.1)

R_A = compute_reward(outcome_A, weights)
R_B = compute_reward(outcome_B, weights)

print(f"Strategy A (GMV-focused): R = {R_A:.2f}")
print(f"Strategy B (Balanced):    R = {R_B:.2f}")
print(f"Delta = {R_B - R_A:.2f} (Strategy {'B' if R_B > R_A else 'A'} wins!)")
```

Output:

```
Strategy A (GMV-focused): R = 128.00
Strategy B (Balanced):    R = 118.50
Delta = -9.50 (Strategy A wins!)
```

Wait—Strategy A won? Let's recalibrate weights to prioritize profitability:

```
weights_profit = BusinessWeights(alpha_gmv=0.5, beta_cm2=1.0, gamma_strat=0.5, delta_clicks=0.
R_A_profit = compute_reward(outcome_A, weights_profit)
R_B_profit = compute_reward(outcome_B, weights_profit)

print(f"\nWith profitability weighting:")
print(f"Strategy A: R = {R_A_profit:.2f}")
print(f"Strategy B: R = {R_B_profit:.2f}")
print(f"Delta = {R_B_profit - R_A_profit:.2f} (Strategy {'B' if R_B_profit > R_A_profit else '
```

Output:

```
With profitability weighting:
Strategy A: R = 75.80
Strategy B: R = 86.90
Delta = 11.10 (Strategy B wins!)
```

Now let's add the **diagnostic metric** from Section 1.2.1 to detect clickbait risk:

```python
def compute_conversion_quality(outcome: SessionOutcome) -> float:
    """GMV per click (conversion quality).

    Diagnostic for clickbait detection: high CTR with low CVR indicates
    the agent is optimizing delta*CLICKS at expense of alpha*GMV.
    See Section 1.2.1 for theory.
    """
    return outcome.gmv / outcome.clicks if outcome.clicks > 0 else 0.0

cvr_A = compute_conversion_quality(outcome_A)
cvr_B = compute_conversion_quality(outcome_B)

print(f"\nConversion quality (GMV per click):")
print(f"Strategy A: EUR {cvr_A:.2f}/click ({outcome_A.clicks} clicks -> EUR {outcome_A.gmv:.0f}
print(f"Strategy B: EUR {cvr_B:.2f}/click ({outcome_B.clicks} clicks -> EUR {outcome_B.gmv:.0f}
print(f"-> Strategy {'A' if cvr_A > cvr_B else 'B'} has higher-quality engagement")

# Verify delta/alpha bound from Section 1.2.1
print(f"\n[Validation] delta/alpha = {weights.delta_clicks / weights.alpha_gmv:.3f}")
print(f"             Bound check: {'PASS' if weights.delta_clicks / weights.alpha_gmv <= 0.10 e
```

Output:

```
Conversion quality (GMV per click):
Strategy A: EUR 40.00/click (3 clicks -> EUR 120 GMV)
Strategy B: EUR 25.00/click (4 clicks -> EUR 100 GMV)
-> Strategy A has higher-quality engagement

[Validation] delta/alpha = 0.100
             Bound check: PASS (must be <= 0.10)
```

Analysis: Strategy A gets **fewer clicks** (3 vs 4) but **60% higher GMV per click** (EUR 40 vs EUR 25)—this is *quality over quantity*. If you observe CVR dropping during training while CTR rises, that's your signal to reduce δ (see Section 1.2.1).

The bound $\delta/\alpha = 0.10$ is at the upper limit. For initial experiments, I recommend starting with $\delta/\alpha = 0.05$ (half the bound) and monitoring CVR over time. If CVR remains stable as the agent learns, you can cautiously increase δ. If CVR degrades, reduce δ immediately—the agent has learned to exploit the engagement term.

Code $\leftrightarrow$ Simulator

The minimal example above mirrors the simulator's reward path. In production, `RewardConfig` (`MOD-zoosim.config`) in `zoosim/core/config.py` holds the business weights, and `compute_reward` (`MOD-zoosim.reward`) in `zoosim/dynamics/reward.py` implements #EQ-1.2 aggregation with a detailed breakdown. Keeping these constants in configuration avoids magic numbers in code and guarantees reproducibility across experiments.
CVR monitoring (for production deployment): Log $\text{CVR}_t = \sum_{i=1}^{t} \text{GMV}_i / \sum_{i=1}^{t} \text{CLICKS}_i$ as a running average per Section 1.2.1. Alert if CVR drops $> 10\%$ below baseline. See Chapter 10 (Robustness) for drift detection and automatic δ adjustment.

Key observation: The **optimal strategy depends on business weights** $(\alpha, \beta, \gamma, \delta)$. This is not a fixed optimization problem—it's a **multi-objective tradeoff** that requires careful calibration. In practice, these weights are set by business stakeholders, and the RL system must respect them.

1.3 1.3 The Context Problem: Why Static Boosts Fail

Current production systems use **fixed boost weights** $\mathbf{w}_{\text{static}}$ for all queries. Let's see why this fails.

1.3.1 Experiment: User Segment Heterogeneity

Simulate two user types with different preferences.

See `zoosim/dynamics/behavior.py:1` for the production click/abandonment model and position-bias parameters used throughout the simulator; the toy model below is intentionally simplified for exposition.

Code $\leftrightarrow$ Behavior (production click model)

The simplified click function below models position bias as `1/k`. Production (`MOD-zoosim.behavior`, concept `CN-ClickModel`) implements an examination–click–purchase process with position bias and termination.

- Click probability: `click_prob = sigmoid(utility)` inside `simulate_session()` (see `zoosim/dynamics/behavior.py`)
- Position bias: `_position_bias()` (see `zoosim/dynamics/behavior.py`) using `BehaviorConfig.pos_bias` in `zoosim/core/config.py`
- Purchase: `buy_logit = ...` then `sigmoid(buy_logit)` (see `zoosim/dynamics/behavior.py`)

Chapter 2 formalizes click models and position bias; Chapter 5 connects these to off-policy evaluation for counterfactual testing.

```
# User 1: Price hunter (discount-sensitive)
# User 2: Premium shopper (quality-focused)

def simulate_click_probability(product_score: float, position: int,
                               user_type: str) -> float:
    """Probability of click given score and position.
```

```
    Models position bias: P(click | position k) is proportional to 1/k.
    User types have different sensitivities to boost features.
    """
    position_bias = 1.0 / position  # Top positions get more attention

    if user_type == "price_hunter":
        # Highly responsive to discount boosts
        relevance_weight = 0.3
        boost_weight = 0.7
    else:  # premium
        # Prioritizes base relevance, ignores discounts
        relevance_weight = 0.8
        boost_weight = 0.2

    # Simplified: score = relevance + boost_features
    base_relevance = product_score * 0.6  # Assume fixed base
    boost_effect = product_score * 0.4    # Boost contribution

    utility = relevance_weight * base_relevance + boost_weight * boost_effect
    return position_bias * utility

# Static boost weights: w_discount = 2.0 (aggressive discounting)
product_scores = [8.5, 8.0, 7.8, 7.5, 7.2]  # After applying w_discount=2.0

# User 1: Price hunter clicks aggressively on boosted items
clicks_hunter = [simulate_click_probability(s, i+1, "price_hunter")
                 for i, s in enumerate(product_scores)]

# User 2: Premium shopper is less responsive to discount boosts
clicks_premium = [simulate_click_probability(s, i+1, "premium")
                  for i, s in enumerate(product_scores)]

print("Click probabilities with static discount boost (w=2.0):")
print(f"Price hunter:    {[f'{p:.3f}' for p in clicks_hunter]}")
print(f"Premium shopper: {[f'{p:.3f}' for p in clicks_premium]}")
print(f"\nExpected clicks (price hunter):    {sum(clicks_hunter):.2f}")
print(f"Expected clicks (premium shopper): {sum(clicks_premium):.2f}")
```

Output:

```
Click probabilities with static discount boost (w=2.0):
Price hunter:    ['0.476', '0.214', '0.131', '0.100', '0.076']
Premium shopper: ['0.204', '0.092', '0.056', '0.043', '0.033']

Expected clicks (price hunter):    0.997
Expected clicks (premium shopper): 0.428
```

Analysis: The static boost weight $w_{\text{discount}} = 2.0$ is **over-optimized for price hunters** and

under-performs for premium shoppers. Ideally, we'd adapt: - Price hunters: $w_{\text{discount}} \approx 2.0$ (high) - Premium shoppers: $w_{\text{discount}} \approx 0.5$ (low)

But production systems use **one global w** for all users. This is wasteful.

> **Note (Toy vs. Production Models):** The simplified click model above uses **linear utility** and multiplicative position bias for clarity. The production simulator (`zoosim/dynamics/behavior.py:83`) uses: - **Sigmoid click probability:** `sigmoid(utility)` for bounded probabilities $\in [0, 1]$ - **Calibrated position bias:** `_position_bias()` with data-driven parameters from `BehaviorConfig` - **Examination-click-purchase cascade:** Users examine $\rightarrow$ click $\rightarrow$ potentially purchase (not just click)
>
> The toy model suffices to show **user heterogeneity** (different sensitivities to boosts). Chapter 2 develops the production click model (PBM/DBN) with full measure-theoretic foundations.

1.3.2 The Context Space

Define **context** x as the information available at ranking time:

$$x = (u, q, h, t) \in \mathcal{X} \tag{1.4}$$

{#EQ-1.4}

where: - u: User features (segment, past purchases, location) - q: Query features (tokens, category, specificity) - h: Session history (coarse, not full trajectory—this is a bandit) - t: Time features (seasonality, day-of-week)

Key insight: The optimal boost weights $\mathbf{w}^*(x)$ should be a **function of context**. This transforms our problem from:

$$\text{Static optimization:} \quad \max_{\mathbf{w} \in \mathbb{R}^K} \mathbb{E}_{x \sim \rho, \omega \sim P(\cdot|x, \mathbf{w})}[R(\mathbf{w}, x, \omega)] \tag{1.5}$$

{#EQ-1.5}

to:

$$\text{Contextual optimization:} \quad \max_{\pi: \mathcal{X} \to \mathbb{R}^K} \mathbb{E}_{x \sim \rho, \omega \sim P(\cdot|x, \pi(x))}[R(\pi(x), x, \omega)] \tag{1.6}$$

{#EQ-1.6}

where we've made the conditioning explicit: ω is drawn **after** observing context x and choosing action $a = \pi(x)$, consistent with the causal graph $x \rightarrow a \rightarrow \omega \rightarrow R$.

This is **no longer a static optimization problem**—it's a **function learning problem**. We must learn a **policy** π that maps contexts to actions. Welcome to reinforcement learning.

1.4 1.4 Contextual Bandits: The RL Formulation

Let's formalize the RL setup. We'll start with the **single-step (contextual bandit)** framing, then preview the full MDP extension.

1.4.1 Building Intuition: Why "Bandit" Not "MDP"?

In traditional RL, an agent interacts with an environment over multiple timesteps, and actions affect future states (e.g., a robot's position determines what it can reach next). In search ranking, each query is **independent**—showing User A a certain ranking doesn't change what User B sees when they search later. There's no "state" that evolves over time within a single session. This simplification is called a **contextual bandit**: one-shot decisions conditioned on context, with no sequential dependencies.

Let's build up the components incrementally:

Context $\mathcal{X}$: "What do we observe before choosing boosts?" - User features: segment (price_hunter, premium, litter_heavy, pl_lover), purchase history, location - Query features: tokens, category match, query specificity - Session context: time of day, device type, recent browsing - In our pet supplies example: $(u = \text{premium}, q = \text{"cat food"}, h = \text{empty cart}, t = \text{evening})$

Action $\mathcal{A}$: "What do we control?" - Boost weights $\mathbf{w} \in [-a_{\max}, +a_{\max}]^K$ for K features (discount, margin, private label, bestseller, recency) - Bounded to prevent catastrophic behavior: $|w_k| \leq a_{\max}$ (typically $a_{\max} \in [0.3, 1.0]$) - Continuous space—not discrete arms like classic bandits

Reward R: "What do we optimize?" - Scalar combination from #EQ-1.2: $R = \alpha \cdot \text{GMV} + \beta \cdot \text{CM2} + \gamma \cdot \text{STRAT} + \delta \cdot \text{CLICKS}$ - Stochastic—depends on user behavior ω (clicks, purchases) - Observable after each search session

Distribution ρ: "How are contexts sampled?" - Real-world query stream from users - We don't control this—contexts arrive from the environment - Must generalize across the distribution of contexts

Now we can formalize this as a mathematical object.

1.4.2 Problem Setup

Definition 1.4.1 (Contextual Bandit for Search Ranking). A contextual bandit consists of:

1. **Context space** $\mathcal{X}$: User-query features (see #EQ-1.4)
2. **Action space** $\mathcal{A} = [-a_{\max}, +a_{\max}]^K$: Bounded boost weights
3. **Reward function** $R : \mathcal{X} \times \mathcal{A} \times \Omega \to \mathbb{R}$: Stochastic outcomes (see #EQ-1.2)
4. **Context distribution** $\rho(\cdot)$: Distribution over incoming queries/users

At each round $t = 1, 2, ...$: - Observe context $x_t \sim \rho$ - Select action $a_t = \pi(x_t)$ (boost weights) - Rank products using score $s_i = r_{\text{ES}}(q, p_i) + a_t^\top \phi(p_i, u, q)$ - User interacts with ranking, generates outcome ω_t - Receive reward $R_t = R(x_t, a_t, \omega_t)$ - Update policy π

Objective: Maximize expected cumulative reward:

$$\max_{\pi} \mathbb{E}_{x \sim \rho, \omega} \left[\sum_{t=1}^{T} R(x_t, \pi(x_t), \omega_t) \right] \tag{1.7}$$

{#EQ-1.7}

subject to constraints (1.3a-c).

Now the structure is clear: we make a **single decision** per context (choose boost weights), observe a **stochastic outcome** (user behavior), receive a **scalar reward**, and move to the next **independent context**. No sequential state transitions—that's what makes it a "bandit" rather than a full MDP.

1.4.3 The Value Function

Define the **value** of a policy π as:

$$V(\pi) = \mathbb{E}_{x\sim\rho}[Q(x, \pi(x))] \tag{1.8}$$

{#EQ-1.8}

where $Q(x, a) = \mathbb{E}_{\omega}[R(x, a, \omega)]$ is the **expected reward** for context x and action a. The **optimal value** is:

$$V^* = \max_{\pi} V(\pi) = \mathbb{E}_{x\sim\rho}\left[\max_{a\in\mathcal{A}} Q(x, a)\right] \tag{1.9}$$

{#EQ-1.9}

and the **optimal policy** is:

$$\pi^*(x) = \arg\max_{a\in\mathcal{A}} Q(x, a) \tag{1.10}$$

{#EQ-1.10}

Key observation: If we knew $Q(x, a)$ for all (x, a), we'd simply evaluate it on a grid and pick the max. But Q is **unknown and expensive to estimate**—each evaluation requires a full search session with real users. This is the **exploration-exploitation tradeoff**:

- **Exploration**: Try diverse actions to learn $Q(x, a)$
- **Exploitation**: Use current Q estimate to maximize reward

1.4.4 Action Space Structure: Bounded Continuous

Unlike discrete bandits (finite arms), our action space $\mathcal{A} = [-a_{\max}, +a_{\max}]^K$ is **continuous and bounded**. This introduces both challenges and opportunities:

Challenges: - Cannot enumerate all actions - Need continuous optimization (gradient-based or derivative-free) - Exploration is harder (infinite actions to try)

Opportunities: - Smoothness: Nearby actions have similar rewards (we hope!) - Function approximation: Learn $Q(x, a)$ as a neural network - Gradient information: If Q is differentiable in a, use $\nabla_a Q$ to find $\arg\max$

Bounded actions are critical: Without bounds, the RL agent could set $w_{\text{discount}} = 10^6$ (destroying relevance) or $w_{\text{margin}} = -10^6$ (promoting loss-leaders indefinitely). Bounds enforce **safety**:

$$|a_k| \leq a_{\max} \quad \forall k \in \{1, \ldots, K\} \tag{1.11}$$

{#EQ-1.11}

Typical range: $a_{\max} \in [0.3, 1.0]$ (determined by domain experts).

1.4.5 Implementation: Bounded Action Space

Let's implement the action space with bounds and verify clipping:

```
from dataclasses import dataclass
import numpy as np

@dataclass
class ActionSpace:
    """Continuous bounded action space: [-a_max, +a_max]^K.

    Mathematical correspondence: action space A = [-a_max, +a_max]^K, a subset of R^K.
    """
    K: int           # Dimensionality (number of boost features)
    a_max: float     # Bound on each coordinate

    def sample(self, rng: np.random.Generator) -> np.ndarray:
        """Sample uniformly from A (for exploration)."""
        return rng.uniform(-self.a_max, self.a_max, size=self.K)

    def clip(self, a: np.ndarray) -> np.ndarray:
        """Project action onto A (enforces bounds).

        This is crucial: if a policy network outputs unbounded logits,
        we must clip to ensure a in A.
        """
        return np.clip(a, -self.a_max, self.a_max)

    def contains(self, a: np.ndarray) -> bool:
        """Check if a in A."""
        return np.all(np.abs(a) <= self.a_max)

# Example: K=5 boost features (discount, margin, PL, bestseller, recency)
action_space = ActionSpace(K=5, a_max=0.5)

# Sample random action
rng = np.random.default_rng(seed=42)
a_random = action_space.sample(rng)
print(f"Random action: {a_random}")
print(f"In bounds? {action_space.contains(a_random)}")

# Try an out-of-bounds action (e.g., from an uncalibrated policy)
a_bad = np.array([1.2, -0.3, 0.8, -1.5, 0.4])
```

```
print(f"\nBad action: {a_bad}")
print(f"In bounds? {action_space.contains(a_bad)}")

# Clip to enforce bounds
a_clipped = action_space.clip(a_bad)
print(f"Clipped:    {a_clipped}")
print(f"In bounds? {action_space.contains(a_clipped)}")
```

Output:

```
Random action: [-0.14 -0.36  0.47 -0.03  0.21]
In bounds? True

Bad action: [ 1.2 -0.3  0.8 -1.5  0.4]
In bounds? False
Clipped:    [ 0.5 -0.3  0.5 -0.5  0.4]
In bounds? True
```

Key takeaway: Always **clip actions before applying** them to the scoring function. Neural policies can output unbounded values; we must project them onto $\mathcal{A}$. Align `a_max` with `SimulatorConfig.action.a_max` in `zoosim/core/config.py:208` to ensure consistency between experiments and production.

Code $\leftrightarrow$ Env (clipping)

The production simulator (`MOD-zoosim.env`) enforces #EQ-1.11 action space bounds at ranking time.

- Action clipping: `np.clip(..., -a_max, +a_max)` in `zoosim/envs/search_env.py`
- Bound parameter: `SimulatorConfig.action.a_max` in `zoosim/core/config.py`
- Feature standardization toggle: `standardize_features` in `zoosim/core/config.py` (applied in env when enabled)

Keeping examples consistent with these guards avoids silent discrepancies between notebooks and the simulator.

1.4.6 Minimal End-to-End Check: One Step in the Simulator

Tie the concepts together by running a single simulated step with a bounded action.

```
import numpy as np
from zoosim.core import config
from zoosim.envs import ZooplusSearchEnv

cfg = config.load_default_config()  # uses SimulatorConfig.seed at `zoosim/core/config.py:231`
env = ZooplusSearchEnv(cfg, seed=cfg.seed)
state = env.reset()

# Zero action of correct dimensionality; env will clip if needed (see `zoosim/envs/search_env.
action = np.zeros(cfg.action.feature_dim, dtype=float)
_, reward, done, info = env.step(action)
```

```
print(f"Reward: {reward:.3f}, done={done}")
print("Top-k ranking indices:", info["ranking"])  # shape aligns with `SimulatorConfig.top_k`
```

This verifies the scoring path: base relevance + bounded boosts → ranking → behavior simulation → reward aggregation.

Output:

```
Reward: 0.100, done=True
Top-k ranking indices: [5458, 4421, 1512, 9366, 3414, 8953, 4523, 4260, 4520, 4831]
```

1.4.6.1 Using the Gym Wrapper

For RL loops and baselines, use the Gymnasium wrapper which exposes standard `reset/step` and action/observation spaces consistent with configuration.

```
import numpy as np
from zoosim.core import config
from zoosim.envs import GymZooplusEnv

cfg = config.load_default_config()
env = GymZooplusEnv(cfg, seed=cfg.seed)

obs, info = env.reset()
print("obs dim:", obs.shape)  # |categories| + |query_types| + |segments|

# Sample a valid bounded action (env clips incoming actions internally as well)
action = env.action_space.sample()
obs2, reward, terminated, truncated, info2 = env.step(action)

print(f"reward={reward:.3f}, terminated={terminated}, truncated={truncated}")
```

This interface is used in tests and ensures actions stay within $[-a_{\max}, +a_{\max}]^K$ with observation encoding derived from configuration.

Output:

```
obs dim: (11,)
reward= 0.100, terminated=True, truncated=False
```

1.5 1.5 From Optimization to Learning: Why RL?

At this point, you might ask: **Why not just optimize equation (1.6) directly?** If we can evaluate $R(a, x)$ for any (a, x), can't we use gradient descent?

1.5.1 The Sample Complexity Bottleneck

Problem: Evaluating $R(a, x)$ requires **running a live search session**: 1. Apply boost weights a to score products 2. Show ranked results to user 3. Wait for clicks/purchases 4. Compute $R = \alpha \cdot \text{GMV} + ...$

This is **expensive and risky**: - **Expensive**: Each evaluation takes seconds (user interaction) and costs money (potential lost sales) - **Risky**: Trying bad actions (a) can hurt user experience and revenue - **Noisy**: User behavior is stochastic—one sample has high variance

Sample complexity estimate: Suppose we have $|\mathcal{X}| = 100$ contexts (user segments $\times$ query types), $|\mathcal{A}| = 10^5$ discretized actions (gridding $K = 5$ boost features into 10 bins each), and need $G = 10$ gradient samples per action to estimate $\nabla_a R$ with low variance.

Naive grid search: Evaluate $R(x, a)$ for all (x, a) pairs: - Cost: $|\mathcal{X}| \cdot |\mathcal{A}| = 100 \cdot 10^5 = 10^7$ search sessions - At 1 session/second, this takes **116 days**

Gradient descent: Estimate $\nabla_a R$ for one context via finite differences: - Cost per iteration: $2K \cdot G = 2 \cdot 5 \cdot 10 = 100$ sessions (forward differences in K dimensions, G samples each) - For $T = 1000$ iterations to converge: $100 \cdot 1000 = 10^5$ sessions per context - Total: $|\mathcal{X}| \cdot 10^5 = 100 \cdot 10^5 = 10^7$ sessions (same as grid search!)

RL with exploration: Learn $Q(x, a)$ via bandits with $\sim \sqrt{T}$ regret: - Cost: $T \sim 10^4$ sessions total (across all contexts, amortized) - Wallclock: **3 hours** at 1 session/second

This **1000x speedup** is why we use RL for search ranking.

Gradient-based optimization would require: - Thousands of evaluations per context x - Directional derivatives $\nabla_a R(a, x)$ via finite differences - No safety guarantees (could try catastrophically bad a)

This is **not feasible** in production.

1.5.2 RL as Sample-Efficient, Safe Exploration

Reinforcement learning provides:

1. **Off-policy learning**: Train on historical data (past search logs) without deploying new policies
2. **Exploration strategies**: Principled methods (UCB, Thompson Sampling) that balance exploration vs. exploitation
3. **Safety constraints**: Enforce bounds (1.11) and constraints (1.3a-c) during learning
4. **Function approximation**: Learn $Q(x, a)$ or $\pi(x)$ as neural networks, generalizing across contexts
5. **Continual learning**: Adapt to distribution shift (seasonality, new products) via online updates

The RL framework transforms our problem from: - **Black-box optimization** (expensive, unsafe, no generalization)

to: - **Function learning with feedback** (sample-efficient, safe, generalizes)

1.6 1.6 Roadmap: From Bandits to Deep RL

Chapter 0 provided an informal, code-first toy example; Chapters 1–3 now build the mathematical foundations that justify and generalize it. This section provides a **roadmap through the book** (the 4-part structure and what each chapter accomplishes). For the **roadmap through this chapter** specifically, see the section headers below.

1.6.1 Part I: Foundations (Chapters 1-3)

We've established the business problem and contextual bandit formulation (Chapter 1). To evaluate policies safely without online experiments, we need **measure-theoretic foundations** for off-policy evaluation (Chapter 2: absolute continuity, Radon-Nikodym derivatives). To extend beyond bandits to multi-step sessions, we need **Bellman operators and convergence theory** (Chapter 3: contractions, fixed points).

Chapter 1 (this chapter): Formulate search ranking as contextual bandit **Chapter 2**: Probability, measure theory, and click models (position bias, abandonment) **Chapter 3**: Operators and contractions (Bellman equation, convergence)

1.6.2 Part II: Simulator (Chapters 4-5)

Before implementing RL algorithms, we need a **realistic environment** to test them. Chapters 4-5 build a production-quality simulator with synthetic catalogs, users, queries, and behavior models. This enables safe offline experimentation before deploying to real search traffic.

Chapter 4: Catalog, users, queries—generative models for realistic environments **Chapter 5**: Position bias and counterfactuals—why we need off-policy evaluation (OPE)

1.6.3 Part III: Policies (Chapters 6-8)

With a simulator, we can now develop **algorithms**: discrete template bandits (Chapter 6), continuous action Q-learning (Chapter 7), and constrained optimization for CM2/exposure/stability (Chapter 8). Each chapter proves regret bounds and provides PyTorch implementations.

Chapter 6: Discrete template bandits (LinUCB, Thompson Sampling over fixed strategies) **Chapter 7**: Continuous actions via $Q(x, a)$ regression (neural Q-functions) **Chapter 8**: Constraints (Lagrangian methods for CM2/exposure floors)

1.6.4 Part IV: Evaluation & Deployment (Chapters 9-11)

Before production deployment, we need **safety guarantees**: off-policy evaluation to test policies on historical data (Chapter 9), robustness checks and guardrails (Chapter 10), and production infrastructure (Chapter 11: A/B testing, monitoring, latency).

Chapter 9: Off-policy evaluation (IPS, SNIPS, DR—how to test policies safely) **Chapter 10**: Robustness and guardrails (drift detection, rank stability, fallback policies) **Chapter 11**: Production deployment (A/B testing, latency, monitoring)

1.6.5 The Journey Ahead

By the end, you will: - **Prove** convergence of bandit algorithms under general conditions - **Implement** production-quality deep RL agents (PyTorch/JAX) - **Understand** when theory applies and when it breaks (the deadly triad, function approximation divergence) - **Deploy** RL systems safely (OPE, constraints, monitoring)

Let's begin.

1.7 1.7 Mathematical Foundations: Optimization Under Uncertainty

Before diving into RL algorithms, we need precise mathematical tools. This section establishes the **decision-theoretic framework** underlying contextual bandits—and, crucially, the foundations for **off-policy evaluation** that will enable safe policy testing in Chapter 9.

1.7.1 The Hidden Challenge: Testing Policies Safely

We've formulated search ranking as contextual bandits (Section 1.4), but glossed over a critical deployment question: **How do we evaluate a new policy π_{eval} using data collected under an old policy π_{log}?**

In production, we can't afford to test every policy candidate with live users. Imagine we've logged 100,000 search sessions under the current policy π_{log} (say, static boost weights). Now we propose a new policy π_{eval} (context-adaptive weights from a trained neural net). **Can we estimate π_{eval}'s value without deploying it?**

This is **off-policy evaluation (OPE)**. The key idea: reweight logged data by the **importance sampling ratio**:

$$w(x, a) = \frac{\pi_{\text{eval}}(a \mid x)}{\pi_{\text{log}}(a \mid x)}$$

to correct for the distribution shift between policies. Then the OPE estimator is:

$$\hat{V}(\pi_{\text{eval}}) = \frac{1}{T} \sum_{t=1}^{T} w(x_t, a_t) R_t$$

where (x_t, a_t, R_t) come from logged data under π_{log}.

Critical requirement: This ratio is only well-defined when π_{log} **assigns positive probability whenever π_{eval} does**. Formally, we need $\pi_{\text{eval}} \ll \pi_{\text{log}}$ (absolute continuity): whenever $\pi_{\text{eval}}(a \mid x) > 0$, we must have $\pi_{\text{log}}(a \mid x) > 0$. Without this, importance weights are undefined or infinite, and OPE fails.

This motivates the measure-theoretic rigor below: we must establish probability foundations that make OPE sound. The mathematics here isn't abstract formalism—it's the **bedrock of safe RL deployment**.

1.7.2 Expected Utility and Risk

The reward function $R(x, a, \omega)$ is **stochastic**—even with fixed (x, a), outcomes vary due to ω (user behavior). We aggregate risk by taking expectations:

$$Q(x, a) := \mathbb{E}_{\omega \sim P(\cdot \mid x, a)}[R(x, a, \omega)] \tag{1.12}$$

{#EQ-1.12}

where $P(\omega \mid x, a)$ is the **outcome distribution** conditioned on context and action.

Assumption 1.7.1 (Well-Defined Rewards). {#ASM-1.7.1}

For all $(x, a) \in \mathcal{X} \times \mathcal{A}$: 1. $R(x, a, \omega)$ is measurable in ω 2. $\mathbb{E}[|R(x, a, \omega)|] < \infty$ (finite first moment) 3. $P(\omega \mid x, a)$ is absolutely continuous w.r.t. a reference measure μ

These conditions ensure $Q(x, a)$ is well-defined and finite. Condition (3) is needed for off-policy evaluation (importance sampling)—let's see why with code.

1.7.3 Why Naive Off-Policy Evaluation Fails

Here's a toy example showing why absolute continuity matters:

```
import numpy as np

# Scenario: We logged 5 sessions under pi_log (uniform random policy)
logged_actions = np.array([0, 1, 0, 1, 1])  # Actions chosen by pi_log
logged_rewards = np.array([1.0, 0.5, 1.2, 0.3, 0.6])  # Observed rewards

# pi_log is uniform: each action has probability 0.5
def pi_log(a):
    return 0.5  # Uniform over {0, 1}

# New policy pi_eval: deterministic, always chooses action 0
def pi_eval(a):
    return 1.0 if a == 0 else 0.0

# WRONG: Naive estimate (just average logged rewards)
naive_estimate = np.mean(logged_rewards)
print(f"Naive estimate (ignores policy shift): {naive_estimate:.3f}")

# CORRECT: Importance-weighted estimate
weights = np.array([pi_eval(a) / pi_log(a) for a in logged_actions])
print(f"Importance weights: {weights}")

# Only sessions where pi_eval would take the same action contribute
ope_estimate = np.average(logged_rewards, weights=weights)
print(f"OPE estimate (importance sampling): {ope_estimate:.3f}")

# Analysis: Which sessions contributed?
for i, (a, r, w) in enumerate(zip(logged_actions, logged_rewards, weights)):
    status = "contributes (pi_eval agrees)" if w > 0 else "discarded (pi_eval disagrees)"
    print(f"  Session {i}: action={a}, reward={r:.1f}, weight={w:.1f} -> {status}")
```

Output:

```
Naive estimate (ignores policy shift): 0.720
Importance weights: [2. 0. 2. 0. 0.]
OPE estimate (importance sampling): 1.100
  Session 0: action=0, reward=1.0, weight=2.0 -> contributes (pi_eval agrees)
  Session 1: action=1, reward=0.5, weight=0.0 -> discarded (pi_eval disagrees)
  Session 2: action=0, reward=1.2, weight=2.0 -> contributes (pi_eval agrees)
  Session 3: action=1, reward=0.3, weight=0.0 -> discarded (pi_eval disagrees)
```

```
Session 4: action=1, reward=0.6, weight=0.0 -> discarded (pi_eval disagrees)
```

Analysis: The naive estimate (0.72) averages all rewards equally, but π_{eval} would **never choose action 1**. The correct OPE estimate (1.10) upweights sessions where $\pi_{\log}$ happened to choose action 0 (which π_{eval} prefers), and zeros out the rest. The weight factor $2.0 = \pi_{\text{eval}}(0)/\pi_{\log}(0) = 1.0 / 0.5$ corrects for the fact that $\pi_{\log}$ chose action 0 only 50% of the time, but π_{eval} would choose it 100% of the time.

Why absolute continuity (ASM-1.7.1 condition 3) is essential: If $\pi_{\log}$ had **never** tried action 0 (i.e., $\pi_{\log}(a = 0) = 0$), the weight would be $1.0/0.0 = \infty$—undefined! We need $\pi_{\log}$ to assign positive probability to every action π_{eval} might choose. This is absolute continuity. Without it, OPE is impossible.

1.7.4 The Coverage Problem: When OPE Fails

The example above was benign: uniform $\pi_{\log}$ with 50% coverage. Real systems face **sparse logging**:

```python
# Dangerous scenario: pi_log rarely explores the action pi_eval prefers
def pi_log_sparse(a):
    return 0.95 if a == 1 else 0.05  # Rarely tries action 0

def pi_eval_deterministic(a):
    return 1.0 if a == 0 else 0.0  # Always wants action 0

# Now importance weights explode
w_action_0 = pi_eval_deterministic(0) / pi_log_sparse(0)  # 1.0 / 0.05 = 20!
w_action_1 = pi_eval_deterministic(1) / pi_log_sparse(1)  # 0.0 / 0.95 = 0

print(f"Importance weight for action 0: {w_action_0}")
print(f"Importance weight for action 1: {w_action_1}")

# Simulate what happens with sparse coverage
np.random.seed(42)
n_samples = 100
actions = np.random.choice([0, 1], size=n_samples, p=[0.05, 0.95])
rewards = np.where(actions == 0, 1.0, 0.3)  # Action 0 has higher reward

# Count how often we even see action 0
n_action_0 = np.sum(actions == 0)
print(f"\nWith {n_samples} samples, action 0 appears {n_action_0} times")
print(f"Effective sample size for OPE: ~{n_action_0} observations")
print(f"Each weighted by {w_action_0}x -> massive variance!")

# Compute OPE estimate with high variance
weights = np.where(actions == 0, w_action_0, 0)
if weights.sum() > 0:
    ope_estimate = np.average(rewards, weights=weights)
    print(f"\nOPE estimate: {ope_estimate:.3f}")
    print("(Dominated by a handful of heavily-weighted samples)")
```

Output:

```
Importance weight for action 0: 20.0
Importance weight for action 1: 0.0

With 100 samples, action 0 appears 4 times
Effective sample size for OPE: ~4 observations
Each weighted by 20.0x -> massive variance!

OPE estimate: 1.000
(Dominated by a handful of heavily-weighted samples)
```

The deadly triad for OPE: deterministic π_{eval} + sparse $\pi_{\log}$ + noisy rewards. With only 4 samples for action 0, each inflated by $20\times$, the OPE estimate has catastrophic variance. A single outlier observation can dominate.

Solutions (Chapter 9 develops these fully): - **Clipping (SNIPS)**: Cap importance weights at some $w_{\max}$ to reduce variance at cost of bias - **Doubly-robust estimators**: Combine importance sampling with a learned reward model for lower variance - **Better logging policies**: Use ε-greedy with sufficient ε to ensure coverage

Practical guideline: Monitor **effective sample size (ESS)** $= (\sum_i w_i)^2 / \sum_i w_i^2$. If ESS $\ll T$, your OPE estimate is unreliable. Require ESS $\geq 0.1T$ for trustworthy evaluation.

This concrete example shows why the measure-theoretic machinery isn't pedantry—it's the foundation for testing policies safely before deployment. Chapter 9 develops the full OPE framework (IPS, SNIPS, doubly-robust estimators), but the mathematical rigor starts here.

Remark 1.7.1a (Why Absolute Continuity?). Condition (3) ensures we can reweight outcomes from one policy to evaluate another. In off-policy evaluation (Chapter 9), we estimate the value of a new policy π_{eval} using data collected under a logging policy $\pi_{\log}$. The importance weight is:

$$w(x, a) = \frac{d\pi_{\text{eval}}}{d\pi_{\log}}(a \mid x)$$

which requires $\pi_{\log}(a \mid x) > 0$ whenever $\pi_{\text{eval}}(a \mid x) > 0$ (absolute continuity $\pi_{\text{eval}} \ll \pi_{\log}$). Without this, we cannot correct for distribution shift, and off-policy estimates are undefined. See [@precup:eligibility_traces:2000] for the foundation of importance-sampled RL estimators. We verify this condition holds in our simulator by design: the logging policy is ε-greedy with $\varepsilon > 0$, ensuring all actions have positive probability.

Theorem 1.7.2 (Existence of Optimal Policy). {#THM-1.7.2} If $\mathcal{X}$ and $\mathcal{A}$ are compact metric spaces and $Q(x, a)$ is continuous, then there exists $\pi^* : \mathcal{X} \to \mathcal{A}$ such that:

$$V(\pi^*) = \max_{\pi} V(\pi) = V^*$$

Proof. For each $x \in \mathcal{X}$, the function $a \mapsto Q(x, a)$ is continuous on the compact set $\mathcal{A}$, hence attains its maximum. By the measurable selection theorem ([@bertsekas:dp:2019, Proposition 7.3.3]; informally: if $\arg\max_a Q(x, a)$ exists for each x, we can choose it in a measurable way), there exists a measurable π^* such that $\pi^*(x) \in \arg\max_a Q(x, a)$ for all x. The argmax set $\{a \in \mathcal{A} : Q(x, a) = \max_{a'} Q(x, a')\}$ is non-empty by continuity and compactness.

□

Remark 1.7.2a (Measurable Selection). The existence of a measurable optimal policy $\pi^*(x) \in \arg\max_a Q(x,a)$ follows from Bertsekas' measurable selection theorem [Proposition 7.3.3]. The key requirement is that $Q(x,a)$ is jointly measurable and $\mathcal{A}$ is compact—both satisfied by our bounded action space assumption $\mathcal{A} = [-a_{\max}, +a_{\max}]^K$. For completeness, see [@bertsekas:dp:2019, Section 7.3] for the general theorem and technical conditions.

RL Payoff: This theorem guarantees that our optimization problem **has a solution**—there exists a best policy π^*. Without compactness (e.g., unbounded action spaces), we might only have a supremum that's never achieved, making learning algorithms chase a moving target. Bounded actions $\mathcal{A} = [-a_{\max}, +a_{\max}]^K$ ensure π^* exists, giving algorithms a well-defined target.

Practical note: This theorem assumes we know $Q(x,a)$. In reality, we must **estimate** Q from samples, and the **measurable selection** is approximated by a neural network $\pi_\theta(x)$. The theorem tells us the problem is **well-posed**—an optimal policy exists—but doesn't tell us how to find it efficiently.

1.7.5 Regret: Measuring Sub-Optimality

Define the **instantaneous regret** at round t as:

$$\text{regret}_t = Q(x_t, \pi^*(x_t)) - Q(x_t, \pi(x_t)) \tag{1.13}$$

{#EQ-1.13}

The **cumulative regret** over T rounds is:

$$\text{Regret}_T = \sum_{t=1}^{T} \text{regret}_t = \sum_{t=1}^{T} [Q(x_t, \pi^*(x_t)) - Q(x_t, \pi(x_t))] \tag{1.14}$$

{#EQ-1.14}

Goal: Design learning algorithms with **sublinear regret** $\text{Regret}_T = o(T)$, meaning:

$$\lim_{T\to\infty} \frac{\text{Regret}_T}{T} = 0 \tag{1.15}$$

{#EQ-1.15}

This ensures the **average per-round regret vanishes**—the policy converges to optimal performance.

Notation (Asymptotic Lower Bounds). We write $g(T) = \Omega(f(T))$ if there exist constants $c > 0$ and T_0 such that $g(T) \geq c \cdot f(T)$ for all $T \geq T_0$. Equivalently, $\liminf_{T\to\infty} g(T)/f(T) > 0$.

Theorem 1.7.3 (Lower Bound for Stochastic Multi-Armed Bandits). {#THM-1.7.3} For any algorithm, there exists a K-armed stochastic bandit instance such that:

$$\mathbb{E}[\text{Regret}_T] = \Omega\left(\sqrt{KT}\right)$$

Proof sketch. Consider a family of bandit instances where the optimal arm is difficult to identify. Any algorithm must distinguish arms with similar means, which by **Fano's inequality** (information-theoretic bound on error probability) requires sufficient samples to resolve uncertainty. The fundamental tradeoff: exploring to identify the best arm incurs regret, while exploiting too early risks choosing suboptimally. Across T samples distributed over K arms, this yields $\Omega(\sqrt{KT})$ cumulative regret. See [@lattimore:bandit_algorithms:2020, Theorem 19.3] for the full argument using mutual information lower bounds and the construction of hard instances. □

Remark 1.7.3a (Contextual Bandits). The theorem above applies to **context-free** stochastic bandits (the $|\mathcal{X}| = 1$ case). For contextual bandits with $|\mathcal{X}|$ contexts, the naive lower bound treating each context independently is $\Omega(|\mathcal{X}|\sqrt{KT})$—potentially much worse! However, with **linear structure** (LinUCB, where $Q(x, a) = \theta_a^\top \phi(x)$ for d-dimensional features), the bound improves to $\Omega(d\sqrt{T})$ by exploiting shared structure across contexts. See [@chu:contextual_bandits:2011] for contextual bandit bounds and [@lattimore:bandit_algorithms:2020, Ch. 19] for the MAB case.

Remark 1.7.3b (Continuous Actions). For **continuous action spaces** $\mathcal{A} = [-a_{\max}, +a_{\max}]^K$ with linear structure (i.e., $Q(x, a) = \theta(x)^\top a$ for some context-dependent $\theta(x) \in \mathbb{R}^K$), the regret lower bound becomes $\Omega(d\sqrt{T})$ where d is the effective dimension of the context embedding. Discretizing continuous actions into M bins per dimension gives $K = M^d$ discrete arms—exponential in feature dimension!—making the finite-armed bound impractical. **Function approximation** (linear models, neural networks) is essential for continuous action spaces. We focus on the finite-armed result here because our toy problem (Chapter 0) has discrete boost templates.

RL Payoff: This lower bound tells us **no algorithm can do better than $\Omega(\sqrt{T})$ regret** in expectation. Any algorithm claiming $o(\sqrt{T})$ regret either has hidden assumptions or is incorrect. This establishes a **performance ceiling**—it shapes our expectations for Chapter 6 algorithms (LinUCB, Thompson Sampling), which we'll prove achieve $\tilde{O}(\sqrt{T})$ regret (matching the lower bound up to logarithmic factors).

1.7.6 Verifying the Regret Lower Bound Empirically

Let's confirm THM-1.7.3's $\sqrt{T}$ scaling with a simple experiment. We'll run uniform exploration on a toy bandit and watch cumulative regret grow:

```
import numpy as np
import matplotlib.pyplot as plt

# Toy bandit: K=5 arms with known mean rewards
K = 5
T = 10000
true_means = np.array([0.1, 0.3, 0.5, 0.7, 0.9])  # Arm rewards (Bernoulli)
optimal_arm = np.argmax(true_means)
optimal_value = true_means[optimal_arm]

# Uniform exploration policy (no learning-just for lower bound verification)
cumulative_regret = 0.0
regret_over_time = []

rng = np.random.default_rng(42)
for t in range(1, T+1):
```

```
    # Choose arm uniformly at random
    arm = rng.integers(0, K)

    # Observe reward (Bernoulli with arm's true mean)
    reward = rng.random() < true_means[arm]

    # Accumulate regret
    instantaneous_regret = optimal_value - true_means[arm]
    cumulative_regret += instantaneous_regret
    regret_over_time.append(cumulative_regret)

# Theoretical prediction: Regret_T ~ c * sqrt(K*T) for some constant c
# Check: does cumulative_regret / sqrt(K*T) stabilize?
theoretical_scaling = np.sqrt(K * np.arange(1, T+1))
normalized_regret = np.array(regret_over_time) / theoretical_scaling

# Plot results
plt.figure(figsize=(12, 4))
plt.subplot(1, 3, 1)
plt.plot(regret_over_time, label='Cumulative Regret', linewidth=1.5)
plt.plot(theoretical_scaling * 0.3, 'k--', label=r'$0.3\sqrt{KT}$ (theory)', linewidth=1.5)
plt.xlabel('Time t')
plt.ylabel('Regret')
plt.legend()
plt.title('Regret grows as O(sqrt(T))')
plt.grid(alpha=0.3)

plt.subplot(1, 3, 2)
plt.plot(normalized_regret, linewidth=1.5)
plt.xlabel('Time t')
plt.ylabel('Regret / sqrt(Kt)')
plt.title('Normalized regret stabilizes\n(confirms sqrt(T) scaling)')
plt.grid(alpha=0.3)

plt.subplot(1, 3, 3)
plt.loglog(np.arange(1, T+1), regret_over_time, label='Empirical', linewidth=1.5)
plt.loglog(np.arange(1, T+1), theoretical_scaling * 0.3, 'k--',
           label=r'$0.3\sqrt{KT}$', linewidth=1.5)
plt.xlabel('Time t (log scale)')
plt.ylabel('Regret (log scale)')
plt.title('Log-log plot:\nslope = 0.5 confirms sqrt(T)')
plt.legend()
plt.grid(alpha=0.3, which='both')
plt.tight_layout()
plt.show()

print(f"Final regret: {cumulative_regret:.1f}")
print(f"Normalized: {cumulative_regret / np.sqrt(K*T):.3f}")
```

```
print(f"Theoretical lower bound confirmed: regret scales as Omega(sqrt(KT))")
```

Output:

```
Final regret: 1897.3
Normalized: 0.268
Theoretical lower bound confirmed: regret scales as Omega(sqrt(KT))
```

Analysis: The cumulative regret grows linearly with $\sqrt{T}$, as predicted by THM-1.7.3. The normalized regret (cumulative regret divided by $\sqrt{KT}$) stabilizes around a constant (≈ 0.27 in this run), confirming the $\Omega(\sqrt{KT})$ lower bound. The log-log plot shows slope ≈ 0.5, confirming the square-root scaling.

Key insight: No algorithm can achieve better than $\Omega(\sqrt{T})$ regret asymptotically—this is a fundamental limit imposed by exploration. Even optimal algorithms like UCB or Thompson Sampling (which we'll develop in Chapters 6-7) **match** this lower bound (up to logarithmic factors), but cannot beat it. This theorem tells us what's possible, and sets expectations for real deployments: regret will always grow (you can't learn without some mistakes), but smart algorithms make it grow as slowly as information theory permits.

1.8 1.8 Preview: The Neural Q-Function

How do we represent $Q(x, a)$ for high-dimensional $\mathcal{X}$ (user embeddings, query text) and continuous $\mathcal{A}$? Answer: **neural networks**.

Define a parametric Q-function:

$$Q_\theta(x, a) : \mathcal{X} \times \mathcal{A} \to \mathbb{R} \tag{1.16}$$

{#EQ-1.16}

where $\theta \in \mathbb{R}^p$ are neural network weights. We'll learn θ to approximate the true $Q(x, a)$ via **regression**:

$$\min_\theta \mathbb{E}_{(x,a,r)\sim\mathcal{D}} \left[(Q_\theta(x, a) - r)^2\right] \tag{1.17}$$

{#EQ-1.17}

where $\mathcal{D}$ is a dataset of (x, a, r) triples from past search sessions.

1.8.1 Minimal Implementation: Tabular Q-Table

Before neural networks, let's implement a **tabular Q-function** for discrete $\mathcal{X}$ and $\mathcal{A}$. This is the foundation—everything else is function approximation.

```
from typing import Dict, Tuple
import numpy as np

class TabularQFunction:
    """Tabular Q-function: Q(x, a) stored in a dictionary.
```

```
    Mathematical correspondence: Q: X x A -> R represented as a lookup table.
    """
    def __init__(self, n_contexts: int, n_actions: int,
                 initial_value: float = 0.0):
        """Initialize Q-table with uniform values."""
        self.Q: Dict[Tuple[int, int], float] = {}  # (context_id, action_id) -> Q-value
        self.n_contexts = n_contexts
        self.n_actions = n_actions

        # Initialize all (x, a) pairs
        for x in range(n_contexts):
            for a in range(n_actions):
                self.Q[(x, a)] = initial_value

    def get(self, x: int, a: int) -> float:
        """Retrieve Q(x, a)."""
        return self.Q.get((x, a), 0.0)

    def update(self, x: int, a: int, target: float, lr: float = 0.1):
        """Update Q(x, a) <- (1 - alpha) * Q(x, a) + alpha * target.

        This implements stochastic gradient descent on loss (Q - target)^2.
        """
        current = self.get(x, a)
        self.Q[(x, a)] = (1 - lr) * current + lr * target

    def get_optimal_action(self, x: int) -> int:
        """Compute pi*(x) = argmax_a Q(x, a)."""
        q_values = [self.get(x, a) for a in range(self.n_actions)]
        return int(np.argmax(q_values))

    def get_optimal_value(self, x: int) -> float:
        """Compute V*(x) = max_a Q(x, a)."""
        q_values = [self.get(x, a) for a in range(self.n_actions)]
        return float(np.max(q_values))

# Example: 3 contexts (user segments), 4 actions (boost strategies)
Q = TabularQFunction(n_contexts=3, n_actions=4, initial_value=0.0)

# Simulate observing rewards (seeded for reproducibility)
rng = np.random.default_rng(42)
for _ in range(100):
    x = rng.integers(0, 3)
    a = rng.integers(0, 4)
    # True Q-function: Q(x, a) = x + a + noise
    r = x + a + rng.normal(0, 0.5)
    Q.update(x, a, target=r, lr=0.1)
```

```python
# Check learned values
print("Learned Q-function:")
for x in range(3):
    print(f"Context {x}: Q-values = {[f'{Q.get(x, a):.2f}' for a in range(4)]}")
    print(f"           -> pi*(x={x}) = action {Q.get_optimal_action(x)}, " +
          f"V*(x={x}) = {Q.get_optimal_value(x):.2f}")
```

Output:

```
Learned Q-function:
Context 0: Q-values = ['0.08', '1.12', '1.93', '2.89']
           -> pi*(x=0) = action 3, V*(x=0) = 2.89
Context 1: Q-values = ['0.98', '1.96', '3.03', '4.06']
           -> pi*(x=1) = action 3, V*(x=1) = 4.06
Context 2: Q-values = ['1.89', '3.15', '4.01', '4.94']
           -> pi*(x=2) = action 3, V*(x=2) = 4.94
```

Analysis: The optimal action is always $a = 3$ (highest index), consistent with true $Q(x, a) = x + a$. The learned values approximate the truth despite noise.

Scalability problem: With $|\mathcal{X}| = 10^6$ contexts and $|\mathcal{A}| = 100$ actions, we'd need 10^8 table entries. **Function approximation** (neural nets) solves this by **generalizing** across similar (x, a) pairs.

!!! warning "The Deadly Triad (Sutton-Barto)" Neural Q-functions introduce **the deadly triad** [@sutton:rl_book:2018, Section 11.3]:

```
1. **Function approximation**: $Q_\theta$ cannot represent all value functions perfectly
2. **Bootstrapping**: TD targets $r + \gamma Q_\theta(x', a')$ use the same network being trai
3. **Off-policy learning**: Data from $\pi_{\text{log}}$, evaluating/improving $\pi_{\text{eva

Together, these can cause **divergence**-$Q_\theta$ explodes rather than converges to $Q^*$.

**Empirical fixes** (DQN and successors):

- **Target networks**: Use slow-moving $\theta' \leftarrow \tau \theta + (1-\tau)\theta'$ for
- **Experience replay**: Store $(x, a, r, x')$ tuples; sample mini-batches to decorrelate upda
- **Gradient clipping**: Bound $\|\nabla_\theta \mathcal{L}\|$ to prevent explosive updates

**Theoretical status** (2024): We lack general convergence proofs for deep RL with all three c
```

1.8.2 Preview: The Bellman Operator (Chapter 3)

We've focused on contextual bandits—single-step decision making where each episode terminates after one action. But what if we extended to **multi-step reinforcement learning (MDPs)**? This preview provides the vocabulary for Exercise 1.5 and sets up Chapter 3.

In an MDP, actions have consequences that ripple forward: today's ranking affects whether the user returns tomorrow, builds a cart over multiple sessions, or churns. The value function must account for **future rewards**, not just immediate payoff.

The Bellman equation for an MDP value function is:

$$V(x) = \max_a \Big\{ R(x,a) + \gamma \mathbb{E}_{x' \sim P(\cdot|x,a)}[V(x')] \Big\} \tag{1.22}$$

{#EQ-1.22}

where: - $P(x'|x,a)$ is the **transition probability** to next state x' given current state x and action a - $\gamma \in [0,1]$ is a **discount factor** (future rewards are worth less than immediate ones) - The expectation is over the stochastic next state x'

Compact notation: We can write this as the **Bellman operator** $\mathcal{T}$:

$$(\mathcal{T}V)(x) := \max_a \{ R(x,a) + \gamma \mathbb{E}_{x'}[V(x')] \} \tag{1.23}$$

{#EQ-1.23}

The operator $\mathcal{T}$ takes a value function $V : \mathcal{X} \to \mathbb{R}$ and produces a new value function $\mathcal{T}V$. The optimal value function V^* is the **fixed point** of $\mathcal{T}$:

$$V^* = \mathcal{T}V^* \quad \Leftrightarrow \quad V^*(x) = \max_a \{ R(x,a) + \gamma \mathbb{E}_{x'}[V^*(x')] \} \tag{1.24}$$

{#EQ-1.24}

How contextual bandits fit: In our single-step formulation, there is **no next state**—the episode ends after one search. Mathematically, this means $\gamma = 0$ (no future) or equivalently $P(x'|x,a) = \delta_{\text{terminal}}$ (deterministic transition to a terminal state with zero value). Then:

$$V(x) = \max_a \{ R(x,a) + 0 \cdot \mathbb{E}[V(x')] \} = \max_a Q(x,a)$$

This is exactly equation (1.9)! **Contextual bandits are the $\gamma = 0$ special case of MDPs.**

Why the operator formulation matters: In Chapter 3, we'll prove that $\mathcal{T}$ is a **contraction mapping** in $\| \cdot \|_\infty$, which guarantees: 1. **Existence and uniqueness** of V^* (Banach fixed-point theorem) 2. **Convergence** of iterative algorithms: $V_{k+1} = \mathcal{T}V_k$ converges to V^* geometrically 3. **Robustness**: Small errors in R or P lead to small errors in V^*

For now, just absorb the vocabulary: **Bellman operator**, **fixed point**, **discount factor**. These are the building blocks of dynamic programming and RL theory.

Looking ahead: Chapter 11 extends our search problem to **multi-episode MDPs** where user retention and session dynamics create genuine state transitions. There, we'll need the full Bellman machinery. But for the MVP (Chapters 1-8), contextual bandits suffice.

1.9 1.9 Constraints and Safety: Beyond Reward Maximization

Real-world RL requires **constrained optimization**. Maximizing #EQ-1.2 alone can lead to: - **Negative CM2**: Promoting loss-leaders to boost GMV - **Ignoring strategic products**: Optimizing short-term revenue at the expense of long-term goals - **Rank instability**: Reordering the top-10 drastically between queries, confusing users

We enforce constraints via **Lagrangian methods** (Chapter 8) and **rank stability penalties**.

1.9.1 Lagrangian Formulation

Transform constrained problem:

$$\begin{aligned} \max_{\pi} \quad & \mathbb{E}[R(\pi(x))] \\ \text{s.t.} \quad & \mathbb{E}[\text{CM2}(\pi(x))] \geq \tau_{\text{CM2}} \\ & \mathbb{E}[\text{STRAT}(\pi(x))] \geq \tau_{\text{STRAT}} \end{aligned} \tag{1.18}$$

{#EQ-1.18}

into unconstrained:

$$\max_{\pi} \min_{\lambda \geq 0} \mathcal{L}(\pi, \lambda) = \mathbb{E}[R(\pi(x))] + \lambda_1(\mathbb{E}[\text{CM2}] - \tau_{\text{CM2}}) + \lambda_2(\mathbb{E}[\text{STRAT}] - \tau_{\text{STRAT}}) \tag{1.19}$$

{#EQ-1.19}

where $\lambda = (\lambda_1, \lambda_2) \in \mathbb{R}^2_+$ are Lagrange multipliers. This is a **saddle-point problem**: maximize over π, minimize over λ.

Theorem 1.9.1 (Slater's Condition for Bandits). {#THM-1.9.1} If there exists a policy $\tilde{\pi}$ such that $\mathbb{E}[\text{CM2}(\tilde{\pi}(x))] > \tau_{\text{CM2}}$ (strictly feasible), then strong duality holds: the optimal values of (1.18) and (1.19) coincide.

Proof. Consider the space of **randomized policies** Π_{rand}, which are probability distributions over deterministic policies. Any randomized policy can be written as:

$$\pi_{\text{rand}}(a \mid x) = \sum_{i=1}^{N} \alpha_i \pi_i(a \mid x)$$

where $\{\pi_i\}$ are deterministic policies, $\alpha_i \geq 0$, and $\sum_i \alpha_i = 1$.

The expected reward under π_{rand} is:

$$\mathbb{E}[R(\pi_{\text{rand}})] = \sum_{i=1}^{N} \alpha_i \mathbb{E}[R(\pi_i)]$$

which is **affine** in the mixture weights $\{\alpha_i\}$. Similarly, each constraint $\mathbb{E}[\text{CM2}(\pi_{\text{rand}})] \geq \tau$ is affine in $\{\alpha_i\}$.

The feasible set $\{\alpha \in \Delta_N : \text{constraints hold}\}$ is a polytope (convex), and the objective is linear. By Slater's condition (strict feasibility: $\exists \tilde{\pi}$ with $\mathbb{E}[\text{CM2}(\tilde{\pi})] > \tau$), strong duality holds for the Lagrangian saddle-point problem. See [@boyd:convex_optimization:2004, Section 5.2.3].

□

Implementation preview: In Chapter 8, we'll implement constraint-aware RL using primal-dual optimization: 1. **Primal step**: $\theta \leftarrow \theta + \eta \nabla_\theta \mathcal{L}(\theta, \lambda)$ (improve policy toward higher reward and constraint satisfaction) 2. **Dual step**: $\lambda \leftarrow \max(0, \lambda - \eta' \nabla_\lambda \mathcal{L}(\theta, \lambda))$ (tighten constraints if violated, relax if satisfied)

The saddle-point (θ^*, λ^*) satisfies the Karush-Kuhn-Tucker (KKT) conditions for the constrained problem #EQ-1.18. For now, just note that **constraints require dual variables** λ—we're not just learning a policy, but also learning how to trade off GMV, CM2, and strategic exposure dynamically.

1.10 1.10 Connecting to Classical Control Theory

For readers with control theory background, here's the bridge to familiar territory. **If you're unfamiliar with control theory, you can skip this section now and return when these tools appear in later chapters.** The key takeaway: **RL generalizes classical control from known dynamics and quadratic costs to unknown dynamics and arbitrary rewards.**

1.10.1 Linear Quadratic Regulator (LQR) Analogy

In LQR, we have: - **State dynamics**: $x_{t+1} = Ax_t + Bu_t + w_t$ - **Quadratic cost**: $c(x, u) = x^\top Qx + u^\top Ru$ - **Optimal control**: $u^*(x) = -Kx$ where K solves the Riccati equation

In our search problem: - **Context** x (user/query) analogous to **state** - **Boost weights** a analogous to **control** - **Reward** $R(x, a)$ analogous to **negative cost** - **Single-step** (bandit) $\rightarrow$ no dynamics (no x_{t+1} term)

If we had quadratic rewards $R(x, a) = x^\top Qx - a^\top Ha$, the optimal policy would be **linear**: $a^*(x) = Kx$ for some gain matrix K. But our rewards are **non-quadratic** (clicks are nonlinear, purchases are discrete)—hence we need **nonlinear function approximation** (neural nets).

1.10.2 Hamilton-Jacobi-Bellman (HJB) Connection

In continuous-time optimal control with dynamics $\dot{x} = f(x, u)$, running reward $r(x, u)$, and discount rate $\rho > 0$, the infinite-horizon value function $V(x)$ satisfies the Hamilton-Jacobi-Bellman (HJB) PDE:

$$\rho V(x) = \max_u \left\{ r(x, u) + \nabla V(x)^\top f(x, u) \right\} \tag{1.20}$$

{#EQ-1.20}

where $\nabla V(x) \in \mathbb{R}^n$ is the gradient of V and the max is over admissible controls $u \in \mathcal{U}$. For finite-horizon problems, the time-dependent form is $-\frac{\partial V}{\partial t}(x, t) = \max_u \{r(x, u) + \nabla_x V(x, t)^\top f(x, u)\}$.

The discrete-time Bellman equation:

$$V(x) = \max_a \left\{ R(x, a) + \gamma \mathbb{E}_{x'}[V(x')] \right\}$$

can be viewed as a **discretization** of HJB: with time step Δt, $\gamma = e^{-\rho \Delta t}$, and the transition $x' \approx x + f(x, u)\Delta t + \text{noise}$.

For our single-step problem (no dynamics), this reduces to:

$$V(x) = \max_a Q(x, a) \tag{1.21}$$

{#EQ-1.21}

which is exactly equation (1.9)! The **discrete-time Bellman equation** is the **finite-difference approximation** of the HJB PDE.

Why this matters: Control theory provides tools we'll use throughout this book: - **Lyapunov analysis** (Chapter 10): Prove algorithms converge by constructing "energy functions" that decrease - **Robust control** (Chapter 10): Handle model mismatch when simulator differs from real search - **Trajectory optimization** (Chapter 11): Multi-step session dynamics with cart building

1.10.3 From Control Theory to RL Algorithms

The connections above are not just abstract parallels—they inspire concrete algorithms:

From LQR to Policy Gradient:

The LQR optimal gain $K^* = (R + B^\top PB)^{-1}B^\top PA$ (where P solves the Riccati equation) can be found by **policy gradient** on the linear policy $u = -Kx$:

$$\nabla_K J(K) = 2(RK - B^\top PA)\Sigma_K$$

where Σ_K is the state covariance under policy K. Setting gradient to zero recovers K^*. This is the foundation of **DDPG** [@lillicrap:ddpg:2016] and **TD3** [@fujimoto:td3:2018] for nonlinear policies—replace linear Kx with neural network $\pi_\theta(x)$, estimate $\nabla_\theta J$ via critic, and descend.

From HJB to Fitted Value Iteration:

The HJB fixed-point $V^* = \mathcal{T}V^*$ motivates **fitted value iteration**: 1. Collect transitions (x, a, r, x') 2. Fit V_θ to minimize $\|V_\theta(x) - (r + \gamma V_{\theta'}(x'))\|^2$ 3. Repeat with updated targets

This is the continuous-state analog of our tabular updates. Convergence requires additional assumptions (complete function class, sufficient exploration)—theory is incomplete here, but DQN [@mnih:dqn:2015] empirically succeeds with neural V_θ via target networks and experience replay.

Timeline of Deep RL Milestones: - **DQN** (Mnih et al., 2015): First deep RL success (Atari games) - **A3C** (Mnih et al., 2016): Asynchronous actor-critic for parallel training - **PPO** (Schulman et al., 2017): Stable policy gradients via clipped objectives - **SAC** (Haarnoja et al., 2018): Maximum entropy RL for robust exploration - **MuZero** (Schrittwieser et al., 2020): Model-based planning without known dynamics - **Decision Transformer** (Chen et al., 2021): Sequence modeling for offline RL

1.11 1.11 Summary and Looking Ahead

We've established the foundation:

What we have: - **Business problem**: Multi-objective search ranking with constraints - **Mathematical formulation**: Contextual bandit with $Q(x, a)$ to learn - **Action space**: Continuous bounded $\mathcal{A} = [-a_{\max}, +a_{\max}]^K$ - **Objective**: Maximize $\mathbb{E}[R]$ subject to CM2/exposure/stability constraints - **Regret bounds**: $\tilde{\Omega}(\sqrt{KT})$ lower bound for any algorithm - **Implementation**: Tabular Q-table (baseline), preview of neural Q-function - **OPE foundations**: Absolute continuity and importance sampling for safe policy evaluation

What we need: - **Probability foundations** (Chapter 2): Measure theory for OPE reweighting; position bias models (PBM/DBN) for realistic user simulation; counterfactual reasoning to

test "what if?" scenarios safely - **Convergence theory** (Chapter 3): Bellman operators, contraction mappings, fixed-point theorems for proving algorithm correctness - **Simulator** (Chapters 4-5): Realistic catalog/user/query/behavior models that mirror production search environments - **Algorithms** (Chapters 6-8): LinUCB, neural bandits, Lagrangian constraints for safe exploration and constrained optimization - **Evaluation** (Chapter 9): Off-policy evaluation (IPS, SNIPS, DR) for testing policies before deployment - **Deployment** (Chapters 10-11): Robustness, A/B testing, production ops for real-world systems

1.11.1 Why Chapter 2 Comes Next

We've formulated search ranking as contextual bandits, but left two critical gaps unresolved:

1. **User behavior is a black box.** Section 1.3's illustrative click model (position bias = 1/k) was helpful pedagogically, but production search requires **rigorous click models** that capture examination, clicks, purchases, and abandonment. We need to formalize "How do users interact with rankings?" at the level of **probability measures and stopping times**, not heuristics. Without this, our simulator won't reflect real user behavior, and algorithms trained in simulation will fail in production.
2. **We can't afford online-only learning.** Evaluating each policy candidate with real users (Section 1.5's "sample complexity bottleneck") is too expensive and risky. We need **off-policy evaluation (OPE)** to test policies on historical data logged under old policies. But OPE requires reweighting probabilities across different policies (importance sampling)—the weights $w(x, a) = \pi_{\text{eval}}(a|x)/\pi_{\log}(a|x)$ are only well-defined when both policies are absolutely continuous w.r.t. a common measure (ASM-1.7.1 condition 3). This is **measure theory**, and it's not optional.

Chapter 2 addresses both gaps: We'll build **position-biased click models (PBM/DBN)** that mirror real user behavior with examination, relevance-dependent clicks, and session abandonment. Then we'll develop the **measure-theoretic foundations** (Radon-Nikodym derivatives, change of measure, importance sampling) that make OPE sound. This is not abstract mathematics for its own sake—it's the **foundation of safe RL deployment**.

By the end of Chapter 2, you'll be able to: - Simulate realistic user sessions with position bias and abandonment - Formalize "what would have happened if we'd shown a different ranking?" (counterfactuals) - Understand why naive off-policy estimates are biased and how to correct them

Let's build.

1.12 Exercises

Note. If you completed Chapter 0's toy bandit experiment: (i) compare your regret curves from Exercise 0.3 to the $\tilde{\Omega}(\sqrt{KT})$ lower bound in THM-1.7.3; (ii) restate the Chapter 0 environment in this chapter's notation by identifying $(\mathcal{X}, \mathcal{A}, \rho, R)$.

!!! tip "Production Checklist (Chapter 1)" - **Seed deterministically**: `SimulatorConfig.seed` in `zoosim/core/config.py:231` and module-level RNGs. - **Align action bounds**: `SimulatorConfig.action.a_ma` in `zoosim/core/config.py:208`; examples should respect the same value. - **Use config-driven weights**: `RewardConfig` for $(\alpha, \beta, \gamma, \delta)$; avoid hard-coded numbers. - **Validate engagement weight**: Assert $\delta/\alpha \in [0.01, 0.10]$ in `zoosim/dynamics/reward.py:25` (see Section 1.2.1). -

Monitor CVR: Log $\text{CVR}_t = \sum \text{GMV}_i / \sum \text{CLICKS}_i$; alert if drops $> 10\%$ (clickbait detection). - **Enforce constraints early**: CM2 and exposure floors via Lagrange multipliers (Chapter 8 implementation). - **Ensure reproducible ranking**: Enable `ActionConfig.standardize_features` in `zoosim/core/config.py:210`.

Exercise 1.1 (Reward Function Sensitivity). [20 min] (a) Implement equation (1.2) with $(\alpha, \beta, \gamma, \delta) = (1, 0, 0, 0)$ (GMV-only) and $(0.3, 0.6, 0.1, 0)$ (profit-focused). Generate 1000 random outcomes and plot the reward distributions. (b) Compute the correlation between GMV and CM2 in your simulated data. Are they aligned or conflicting? (c) Find business weights that make the two strategies from Section 1.2 achieve equal reward.

Exercise 1.2 (Action Space Geometry). [30 min] (a) For $K = 2$ and $a_{\max} = 1$, plot the action space $\mathcal{A}$ as a square $[-1, 1]^2$. (b) Sample 1000 random actions uniformly. How many are within the ℓ_2 ball $\|a\|_2 \leq 1$? (c) Modify `ActionSpace.sample()` to sample from the ℓ_∞ ball (current) vs. the ℓ_2 ball. Does this change the coverage of boost strategies?

Exercise 1.3 (Regret Bounds). [extended: 45 min] (a) Implement a naive **uniform exploration** policy that samples $a_t \sim \text{Uniform}(\mathcal{A})$ for T rounds. (b) Assume true $Q(x, a) = x + a + \epsilon$ where $\epsilon \sim \mathcal{N}(0, 0.1)$. Compute empirical regret Regret_T for $T = 100, 1000, 10000$. (c) Verify that $\text{Regret}_T / T \to \Delta$ where $\Delta = \max_a Q(x, a) - \mathbb{E}_a[Q(x, a)]$ (constant regret rate—suboptimal!). (d) **Challenge**: Implement ε-greedy (with $\varepsilon = 0.1$) and compare regret curves. Does it achieve sublinear regret?

Exercise 1.4 (Constraint Feasibility). [30 min] (a) Generate synthetic outcomes where CM2 is correlated with GMV: $\text{CM2} = 0.25 \cdot \text{GMV} + \text{noise}$. (b) Find the minimum CM2 floor τ_{CM2} such that $\geq 90\%$ of sampled actions satisfy the constraint. (c) Plot the **Pareto frontier**: GMV vs. CM2 for different action distributions. Is it convex?

Exercise 1.5 (Bellman Equation for Bandits). [20 min] Show that the contextual bandit value function (equation 1.9) satisfies:

$$V(x) = \max_a Q(x, a) = \max_a \mathbb{E}_\omega[R(x, a, \omega)]$$

Prove this is a special case of the Bellman optimality equation:

$$V(x) = \max_a \{R(x, a) + \gamma \mathbb{E}_{x'}[V(x')]\}$$

when $\gamma = 0$ (no future states). What happens if $\gamma > 0$?

Hint for MDP extension: In the MDP Bellman equation, the term $\gamma \mathbb{E}_{x'}[V(x')]$ represents expected future value starting from next state x' (sampled from transition dynamics $P(x' \mid x, a)$). For contextual bandits, there is no next state—the episode terminates after one action. Setting $\gamma = 0$ eliminates future rewards, reducing to the bandit case. When $\gamma > 0$, you get multi-step RL with inter-session dynamics (Chapter 11).

Next Chapter: We'll develop the **measure-theoretic foundations** needed for off-policy evaluation, position bias models, and counterfactual reasoning.