Contents

1 Chapter 0 — Lab Solutions

Vlad Prytula

These solutions demonstrate how theory meets practice in reinforcement learning. Every solution weaves mathematical analysis with runnable code, following the Application Mode principle: **mathematics and code in constant dialogue**.

All outputs shown are actual results from running the code with the specified seeds.

1.1 Lab 0.1 — Tabular Boost Search (Toy World)

Goal: Reproduce the ≥ 90% of oracle guarantee using `scripts/ch00/toy_problem_solution.py`.

1.1.1 Solution

```python
from scripts.ch00.toy_problem_solution import (
    TabularQLearning,
    discretize_action_space,
    run_learning_experiment,
    evaluate_policy,
    OraclePolicy,
)

# Configure experiment (parameters from exercises_labs.md)
actions = discretize_action_space(n_bins=5, a_min=-1.0, a_max=1.0)
print(f"Action space: {len(actions)} discrete templates (5x5 grid)")

agent = TabularQLearning(
    actions,
    epsilon_init=0.9,
    epsilon_decay=0.995,
    epsilon_min=0.05,
    learning_rate=0.15,
)

results = run_learning_experiment(
    agent,
    n_train=800,
    eval_interval=40,
    n_eval=120,
    seed=314,
)

# Compute oracle baseline
oracle = OraclePolicy(actions, n_eval=200, seed=314)
oracle_results = evaluate_policy(oracle, n_episodes=300, seed=314)
oracle_mean = oracle_results['mean_reward']

pct_oracle = 100 * results['final_mean'] / oracle_mean
print(f"Final mean reward: {results['final_mean']:.2f} (target >= 0.90 * oracle)")
print(f"Per-user reward: {results['final_per_user']}")
```

Actual Output:

```
Action space: 25 discrete templates (5x5 grid)
Oracle: Computing optimal actions via grid search...
  price_hunter: w*=(0.5, 0.0), Q*=17.80
  premium: w*=(-1.0, 1.0), Q*=19.02
```

```
  bulk_buyer: w*=(0.5, 1.0), Q*=12.88

Final mean reward: 16.13 (target >= 0.90 * oracle)
Per-user reward: {'price_hunter': 14.62, 'premium': 22.65, 'bulk_buyer': 11.02}

Oracle mean: 16.77
Percentage of oracle: 96.2%
Result: SUCCESS
```

1.1.2 Analysis

1. We achieve 96.2% of oracle performance—well above the 90% target.

The Q-learning agent learns effective context-adaptive policies purely from interaction data.

2. Per-User Performance Breakdown:

Segment	Q-Learning	Oracle	% of Optimal
price_hunter	14.62	17.80*	82.1%
premium	22.65	19.02*	119.0%**
bulk_buyer	11.02	12.88*	85.6%

*Oracle Q-values are per-action estimates; actual oracle mean across users is 16.77.

Why does premium exceed oracle estimates? The stochasticity in user interactions means different random seeds produce different outcomes. The agent found an action that happened to perform well on the evaluation seed.

3. Learned Policy vs. Oracle Policy:

User Type	Learned Action	Oracle Action
price_hunter	(0.5, 0.5)	(0.5, 0.0)
premium	(-1.0, 0.0)	(-1.0, 1.0)
bulk_buyer	(-0.5, 0.5)	(0.5, 1.0)

The learned actions differ from oracle because: 1. Q-table hasn't converged perfectly in 800 episodes 2. Stochastic rewards mean multiple actions have similar expected values 3. The 5×5 grid may not include the truly optimal continuous action

Key Insight: Even without matching the oracle's exact actions, Q-learning achieves near-oracle reward. This demonstrates the robustness of value-based learning.

1.2 Exercise 0.2 (from exercises_labs.md) — Stress-Testing Reward Weights

Goal: Validate that oversized engagement weight δ inflates rewards despite unchanged GMV.

1.2.1 Solution

```
from scripts.ch00.toy_problem_solution import (
    USER_TYPES, compute_reward, rank_products, simulate_user_interaction,
)

# Exact parameters from exercises_labs.md
alpha, beta, delta = 0.6, 0.3, 0.3  # delta intentionally oversized
user = USER_TYPES["price_hunter"]
ranking = rank_products(0.8, 0.1)
interaction = simulate_user_interaction(user, ranking, seed=7)
reward = compute_reward(interaction, alpha=alpha, beta=beta, gamma=0.0, delta=delta)
print(f"Reward with delta={delta:.1f}: {reward:.2f}")
```

Actual Output:

```
Interaction: {'clicks': [5], 'purchases': [], 'n_clicks': 1, 'n_purchases': 0, 'gmv': 0.0, 'cm
Reward with delta=0.3: 0.30
```

This matches the expected output in `exercises_labs.md`.

1.2.2 Extended Analysis

Running 100 samples per user type with a "clickbait" ranking (high discount boost):

```
Standard weights (alpha=0.6, beta=0.3, delta=0.1):
  Ratio delta/alpha = 0.167
  price_hunter   : R=20.01, GMV=28.30, CM2=9.60, Clicks=1.56
  premium        : R=12.71, GMV=17.98, CM2=6.17, Clicks=0.76
  bulk_buyer     : R=11.35, GMV=16.19, CM2=5.12, Clicks=1.01

Oversized delta (alpha=0.6, beta=0.3, delta=0.3):
  Ratio delta/alpha = 0.500 (5x above guideline!)
  price_hunter   : R=20.32 (+1.6%), GMV=28.30, Clicks=1.56  <-- Same GMV, higher reward!
  premium        : R=12.86 (+1.2%), GMV=17.98, Clicks=0.76
  bulk_buyer     : R=11.55 (+1.8%), GMV=16.19, Clicks=1.01
```

Key Observation: With $\delta/\alpha = 0.50$, reward increases $\sim 1.5\%$ while GMV stays constant. The agent could learn to prioritize clicks over conversions—a form of engagement gaming.

Guideline: Keep $\delta/\alpha \leq 0.10$ to ensure GMV dominates the reward signal.

1.3 Exercise 0.1 — Reward Sensitivity Analysis

Goal: Compare learned policies under three reward configurations.

1.3.1 Solution

```
# Three configurations as specified in the exercise
configs = [
    ((1.0, 0.0, 0.0), "Pure GMV"),
```

```
    ((0.4, 0.5, 0.1), "Profit-focused"),
    ((0.5, 0.2, 0.3), "Engagement-heavy"),
]

# Run Q-learning for each configuration
for weights, label in configs:
    result = run_sensitivity_experiment(weights, label, n_train=1200, seed=42)
    print(f"{label} (alpha={weights[0]}, beta={weights[1]}, delta={weights[2]}):")
    print(f"  Learned policy: ...")
```

Actual Output:

```
Pure GMV (alpha=1.0, beta=0.0, delta=0.0):
  Final reward: 23.99
  Final GMV: 23.99
  Learned policy:
    price_hunter    -> w_discount=+1.0, w_quality=+0.0
    premium         -> w_discount=-1.0, w_quality=+1.0
    bulk_buyer      -> w_discount=+0.5, w_quality=+1.0

Profit-focused (alpha=0.4, beta=0.5, delta=0.1):
  Final reward: 14.03
  Final GMV: 24.24
  Learned policy:
    price_hunter    -> w_discount=+0.5, w_quality=+0.0
    premium         -> w_discount=-1.0, w_quality=+1.0
    bulk_buyer      -> w_discount=+0.5, w_quality=+1.0

Engagement-heavy (alpha=0.5, beta=0.2, delta=0.3):
  Final reward: 14.40
  Final GMV: 24.48
  Learned policy:
    price_hunter    -> w_discount=+1.0, w_quality=-1.0
    premium         -> w_discount=-1.0, w_quality=+1.0
    bulk_buyer      -> w_discount=+0.5, w_quality=+1.0
```

1.3.2 Analysis

Does the policy change? Yes, for some user types:

Configuration	price_hunter	premium	bulk_buyer
Pure GMV	$(+1.0, 0.0)$	$(-1.0, +1.0)$	$(+0.5, +1.0)$
Profit-focused	$(+0.5, 0.0)$	$(-1.0, +1.0)$	$(+0.5, +1.0)$
Engagement-heavy	$(+1.0, -1.0)$	$(-1.0, +1.0)$	$(+0.5, +1.0)$

Key Observations:

1. **premium users:** Policy is stable across all configurations at $(-1.0, +1.0)$—quality-heavy.

This makes sense: premium users convert well on quality products regardless of reward weighting.

2. **price_hunter:** Shows the most variation:
   - Pure GMV: $(+1.0, 0.0)$ — maximize discount, ignore quality
   - Profit-focused: $(+0.5, 0.0)$ — moderate discount (high-margin products)
   - Engagement-heavy: $(+1.0, -1.0)$ — extreme discount, actively penalize quality (clickbait risk!)
3. **bulk_buyer:** Stable at $(+0.5, +1.0)$—balanced approach works across configurations.

The engagement-heavy case shows clickbait risk: For price_hunter, the policy shifts to $(+1.0, -1.0)$, actively demoting quality products. This maximizes clicks but may hurt long-term user satisfaction.

1.4 Exercise 0.2 — Action Geometry and the Cold Start Problem

Goal: Understand how exploration strategy effectiveness depends on policy quality.

This exercise teaches a fundamental insight through a structured investigation. We start with a hypothesis, test it empirically, discover it's wrong, diagnose why, and then design an experiment that validates when the original intuition *does* hold.

1.4.1 Part A — The Hypothesis

Intuition suggests that **local exploration**—small perturbations around our current best action—should be more efficient than **uniform random sampling**. After all, once we find a good region of action space, why waste samples exploring far away?

```
# Two exploration strategies:
# - Uniform: When exploring (with prob epsilon), sample ANY action uniformly
# - Local:   When exploring (with prob epsilon), sample NEIGHBORS of current best (+/-1 grid c
```

Hypothesis: Local exploration converges faster because it exploits structure near good solutions (gradient descent intuition).

Let's test this.

1.4.2 Part B — Cold Start Experiment

We train both agents from scratch (Q=0 everywhere) for 500 episodes.

```
from scripts.ch00.lab_solutions import exercise_0_2_action_geometry

results = exercise_0_2_action_geometry(
    n_episodes_cold=500,
    n_episodes_warmup=200,
    n_episodes_refine=300,
```

```
    n_runs=5,
    seed=42,
)
```

Actual Output (Cold Start):

```
Starting BOTH agents from random initialization (Q=0 everywhere).
Training each for 500 episodes...

Results (averaged over 5 runs):
  Strategy       Final Reward        Std
  ------------ -------------- ----------
  Uniform                15.66      18.48
  Local                  10.33      15.43

  Winner: Uniform (by 34.0%)

  ** SURPRISE! Uniform exploration wins decisively.
      Our hypothesis was WRONG. But why?
```

1.4.3 Part C — Diagnosis: The Cold Start Problem

Why does local exploration fail from cold start?

The problem is **initialization**. With Q=0 everywhere: - **Uniform agent**: Explores the ENTIRE action space randomly - **Local agent**: Starts at action index 0 (the corner: $w = (-1, -1)$) and only explores NEIGHBORS of that corner!

The local agent is doing **"local refinement of garbage"**—there's no good region nearby to refine. It's stuck in a bad neighborhood.

Action Coverage After 200 Episodes:

```
  Uniform explored 18/25 actions (72%)
  Local explored   4/25 actions (16%)

  Local agent never discovered the optimal region!
```

This is the **COLD START PROBLEM**: > Local exploration assumes you're already in a good basin. > From random initialization, you're not.

1.4.4 Part D — Warm Start Experiment

If local exploration fails from cold start, when SHOULD it work?

Answer: After we've found a good region via global exploration!

Experiment Design: 1. Train with UNIFORM for 200 episodes (find good region) 2. Then continue training with each strategy for 300 more episodes 3. Compare which strategy refines better from this warm start

Actual Output (Warm Start):

```
Results (averaged over 5 runs, after 200 warmup episodes):
  Strategy       Final Reward        Std
  ------------ -------------- ----------
  Uniform               14.04      16.76
  Local                 14.58      17.10

  Winner: Local (by 3.7%)

  Local exploration is now COMPETITIVE (or wins)!
    Once we're in a good basin, local refinement works.
```

Key Observation: The gap between uniform and local *reverses* when starting from a warm policy. From cold start, uniform wins by 34%. From warm start, local actually *wins* by 3.7%! Local exploration works—and even excels—*once you're already in a good region.*

1.4.5 Part E — Synthesis: Adaptive Exploration

The key insight: **EXPLORATION STRATEGY SHOULD ADAPT TO POLICY MATURITY.**

Training Phase	Recommended Strategy	Rationale
Early (cold)	Uniform/global	Find good regions across action space
Late (warm)	Local/refined	Exploit structure within good regions

This is exactly what sophisticated algorithms implement:

Algorithm	Mechanism	Effect
SAC	Entropy bonus $\alpha \cdot H(\pi)$	Encourages broad exploration; decays naturally as policy sharpens
PPO	Decaying entropy coefficient	High entropy early (explore) $\rightarrow$ low late (exploit)
ε-greedy	ε decays ($0.9 \rightarrow 0.05$)	Global early, local late
Boltzmann	Temperature τ decays	High τ = uniform, low τ = local around best

Connection to Theory:

The cold start problem explains why **optimistic initialization** (starting with high Q-values) helps—it forces global exploration before settling into local refinement. Starting with $Q = \infty$ everywhere means the agent must try everything before any action looks "best."

1.4.6 Summary Table

Experiment	Uniform	Local	Winner	Gap
Cold start	15.66	10.33	Uniform	34.0%
Warm start	14.04	14.58	**Local**	3.7%

The same exploration strategy can WIN or LOSE depending on whether the policy is cold (random) or warm (trained). In fact, the winner *reverses*: uniform dominates cold start, but local wins after warm-up!

This is not a bug—it's a fundamental insight about RL exploration.

1.4.7 Practical Guideline

When designing exploration strategies, ask:

> "Is my policy already in a good region?"

- **If no** → Use global/uniform exploration first
- **If yes** → Local refinement is efficient

This principle applies beyond toy examples. In production RL: - **Curriculum learning** starts with easier tasks (warm start for harder ones) - **Transfer learning** initializes from pre-trained policies (warm start) - **Reward shaping** guides early exploration toward good regions

---

1.5 Exercise 0.3 — Regret Analysis

Goal: Track cumulative regret and understand what it tells us about learning.

1.5.1 Background: What Is Regret?

Cumulative regret measures total performance loss compared to an oracle:

$$\text{Regret}(T) = \sum_{t=1}^{T} (R_t^* - R_t)$$

where R_t^* is the oracle's reward and R_t is the agent's reward at episode t.

Sublinear regret means $\text{Regret}(T) = o(T)$, i.e., average regret per episode vanishes:

$$\frac{\text{Regret}(T)}{T} \to 0 \quad \text{as } T \to \infty$$

This confirms the agent is *learning*—eventually performing as well as the oracle.

1.5.2 Solution

```
# Cumulative regret: Regret(T) = Sum_{t=1}^T (R*_t - R_t)
# where R*_t is oracle reward and R_t is agent reward

# Run 2000 episodes with geometric decay: eps_t = 0.9 * 0.998^t
```

Actual Output:

```
=== Exercise 0.3: Regret Analysis ===

Oracle policy computed:
  price_hunter: w*=(1.0, -0.5), Q*=17.46
  premium: w*=(-0.5, 1.0), Q*=19.22
  bulk_buyer: w*=(0.5, 1.0), Q*=12.04

Regret Analysis Summary:
  Total episodes: 2000
  Final cumulative regret: 5680.0
  Average regret per episode: 2.840
  Regret growth slowing? True (avg regret: 6.161 early -> 2.840 late)

Empirical curve fitting (for illustration only):
  sqrt(T) model: Regret(T) ~ 127.0 * sqrt(T)
  Power model: Regret(T) ~ 53.9 * T^0.62

WARNING: Don't conflate empirical fits with asymptotic bounds!
  The exponent alpha=0.62 describes the learning TRANSIENT,
  not a fundamental asymptotic rate.

Theoretical expectations (for reference):
  Constant epsilon-greedy:  Theta(T) -- LINEAR regret (explores forever)
  Geometric decay epsilon:  O(1) -- BOUNDED regret (sum of eps_t converges)
  UCB:                      O(sqrt(KT log T))

Our schedule (eps=0.998^t) is geometric -> regret should plateau eventually.
The T^0.62 fit captures the transient, not the asymptote.
```

1.5.3 Interpretation

Is regret sublinear? Yes. The average regret per episode (2.84) is well below the oracle's mean reward (~16), and inspection of the regret curve shows growth slowing over time.

1.5.4 Theory-Practice Gap: Why Curve Fitting Is Misleading

Caution: Fitting a power law to 2000 points and claiming "regret scales as $O(T^{0.62})$" conflates two different things:

Concept	What It Means
Empirical fit	Regret $\approx c \cdot T^{\alpha}$ for *observed* data
Asymptotic bound	$\lim_{T\to\infty} \text{Regret}(T)/T^{\alpha} < \infty$

The empirical exponent $\alpha = 0.62$ could drift as $T \to \infty$. With finite samples, you can fit almost any functional form.

1.5.5 What Should We Actually Expect?

Our implementation uses **geometric decay**: $\varepsilon_t = 0.9 \cdot 0.998^t$.

This is *summable*:

$$\sum_{t=0}^{\infty} \varepsilon_t = \frac{0.9}{1 - 0.998} = 450$$

Since total exploration is bounded, **regret should plateau** as $T \to \infty$—not grow as any power of T!

Exploration Schedule	Asymptotic Regret	Notes
Constant ε	$\Theta(T)$	Linear! Never stops exploring randomly
$\varepsilon = c/t$	$O(\log T)$ or $O(\sqrt{T})$	Depends on problem structure
$\varepsilon = \varepsilon_0 \cdot \lambda^t$ (geometric)	$O(1)$ — bounded!	Total exploration is finite
UCB	$O(\sqrt{KT \log T})$	Optimal for stochastic bandits

Common misconception: "Constant ε-greedy gives $O(T^{2/3})$." This is **wrong**. Constant ε gives *linear* regret $\Omega(\varepsilon T)$ because you explore forever. The $O(T^{2/3})$ bound requires *decaying* ε or Explore-Then-Commit.

1.5.6 What the Empirical Fit Actually Shows

The $T^{0.62}$ fit over 2000 episodes captures the **transient learning phase**:

1. **Early** ($t < 200$): High $\varepsilon \to$ lots of exploration $\to$ high per-episode regret
2. **Middle** ($200 < t < 1000$): Q-values converging $\to$ regret growth slows
3. **Late** ($t > 1000$): $\varepsilon \approx 0.9 \cdot 0.998^{1000} \approx 0.12 \to$ mostly exploitation

The power-law fit interpolates this transition but doesn't reflect any fundamental asymptotic rate.

1.5.7 The Honest Conclusion

What we can say: - Regret growth slows over time → the agent is learning - Average regret per episode decreases → converging toward oracle performance - With geometric decay, regret will eventually plateau (bounded total regret)

What we should NOT say: - "Regret scales as $O(T^{0.62})$" — this conflates empirical fits with asymptotic bounds - Comparisons to UCB/theoretical bounds without matching assumptions

The key insight: Sublinear regret growth confirms learning. The specific exponent from curve-fitting is an artifact of the learning transient, not a fundamental property.

1.6 Exercise 0.4 — Constrained Q-Learning with CM2 Floor

Goal: Add profitability constraint $\mathbb{E}[\text{CM2} \mid x, a] \geq \tau$ and study the GMV–CM2 tradeoff.

1.6.1 Solution

```python
class ConstrainedQLearning:
    """Q-learning with CM2 floor constraint.

    Maintains separate estimates for Q(x,a) (reward) and CM2(x,a) (margin).
    Filters actions based on estimated CM2 feasibility.
    """
    def get_feasible_actions(self, user_name):
        return [a for a in range(n_actions) if self.CM2[(user_name, a)] >= self.tau]
```

Actual Output:

```
=== Exercise 0.4: Constrained Q-Learning ===

Pareto Frontier (GMV vs CM2):
--------------------------------------------------
tau= 0.0: GMV=23.73, CM2= 7.98, Violations=  0%
tau= 2.0: GMV=23.16, CM2= 8.02, Violations= 50%
tau= 4.0: GMV=22.50, CM2= 7.94, Violations= 50%
tau= 6.0: GMV=21.41, CM2= 6.49, Violations= 72%
tau= 8.0: GMV=23.38, CM2= 8.19, Violations= 58%
tau=10.0: GMV=23.03, CM2= 7.23, Violations= 72%
tau=12.0: GMV=20.17, CM2= 6.93, Violations= 66%

Analysis:
  Unconstrained GMV: 23.73
  Unconstrained CM2: 7.98
  Best CM2 at tau=8.0: CM2=8.19, GMV=23.38
```

1.6.2 Theory-Practice Gap: Per-Episode Constraints Are Hard!

The results don't show a clean Pareto frontier. Why?

1. **High CM2 variance:** CM2 is 0 when no purchase occurs (common!), and can be 30+ when a high-margin product sells. Per-episode CM2 is extremely noisy.
2. **Constraint satisfaction is probabilistic:** Even if $\mathbb{E}[\text{CM2} \mid x, a] \geq \tau$, individual episodes often violate the constraint due to variance.
3. **Optimistic initialization:** We initialize CM2 estimates at 10.0 (optimistic). As estimates converge to true values, many actions become infeasible, leading to policy instability.

Better Approaches (Chapter 3, Section 3.6):

1. **Lagrangian relaxation:** Instead of hard constraints, penalize violations:

$$\max_{\pi} \mathbb{E}[R] - \lambda(\tau - \mathbb{E}[\text{CM2}])$$

2. **Chance constraints:** Require $P(\text{CM2} \geq \tau) \geq 1 - \delta_c$ instead of expected value.
3. **Batch constraints:** Aggregate over episodes/users, not per-episode.

Key Insight: Single-episode CMDP constraints with high-variance outcomes require sophisticated handling. The simple primal feasibility approach shown here is educational but not production-ready.

1.7 Exercise 0.5 — Bandit-Bellman Bridge (Conceptual)

Goal: Show that contextual bandit Q-learning is the $\gamma = 0$ case of MDP Q-learning.

1.7.1 Solution

The Bellman optimality equation:

$$V^*(s) = \max_a \left\{ R(s, a) + \gamma \sum_{s'} P(s' \mid s, a) V^*(s') \right\}$$

Setting $\gamma = 0$:

$$V^*(s) = \max_a R(s, a)$$

The future value term vanishes! The Q-function becomes:

$$Q^*(s, a) = R(s, a) = \mathbb{E}[\text{reward} \mid s, a]$$

This is exactly what our bandit Q-table estimates.

1.7.2 Numerical Verification

```python
def bandit_update(Q, r, alpha):
    """Bandit: Q <- (1-alpha)Q + alpha*r"""
    return (1 - alpha) * Q + alpha * r
```

```python
def mdp_update(Q, r, Q_next_max, alpha, gamma):
    """MDP: Q <- Q + alpha[r + gamma*max(Q') - Q]"""
    td_target = r + gamma * Q_next_max
    return Q + alpha * (td_target - Q)
```

Actual Output:

```
Test 1:
  Initial Q: 5.0, Reward: 7.0, alpha: 0.1
  Bandit update: 5.200000
  MDP update (gamma=0): 5.200000
  Difference: 0.00e+00
  PASSED

Test 2:
  Initial Q: 0.0, Reward: 10.0, alpha: 0.5
  Bandit update: 5.000000
  MDP update (gamma=0): 5.000000
  Difference: 0.00e+00
  PASSED

Test 3:
  Initial Q: -3.0, Reward: 2.0, alpha: 0.2
  Bandit update: -2.000000
  MDP update (gamma=0): -2.000000
  Difference: 4.44e-16  <-- Floating point precision
  PASSED

Test 4:
  Initial Q: 100.0, Reward: 50.0, alpha: 0.01
  Bandit update: 99.500000
  MDP update (gamma=0): 99.500000
  Difference: 0.00e+00
  PASSED

Verified: Bandit Q-update = MDP Q-update with gamma=0
```

1.7.3 Implications

Property	Contextual Bandit	Full MDP
Horizon	1 step	T steps (or infinite)
State transitions	None	$s \to s'$ via $P(s' \mid s, a)$
Update target	r	$r + \gamma \max_{a'} Q(s', a')$
Convergence	Stochastic approximation	Bellman contraction

For Chapter 11 (multi-episode search): Today's ranking affects tomorrow's return probability. This requires $\gamma > 0$ and the full Bellman machinery.

1.8 Summary: Theory–Practice Insights

These labs revealed important insights about RL in practice:

Exercise	Key Discovery	Lesson
Lab 0.1	96.2% of oracle achieved	Q-learning works for small discrete action spaces
Ex 0.1	Policy varies with reward weights	Engagement-heavy configs risk clickbait
Ex 0.2	**Cold start problem discovered**	**Exploration strategy must match policy maturity**
Ex 0.3	Regret growth slows over time	Sublinear regret confirms learning; don't conflate empirical fits with $O(\cdot)$ bounds
Ex 0.4	No clean Pareto frontier	Per-episode constraints need Lagrangian methods
Ex 0.5	Bandit = MDP with $\gamma = 0$	Unified view of bandits and MDPs

Key Lessons:

1. **Q-learning works well** for small discrete action spaces with clear structure
2. **Exploration strategy depends on context**:
   - Cold start → uniform/global exploration
   - Warm start → local refinement is competitive
   - This explains why ε-greedy decays, SAC uses entropy, etc.
3. **Per-episode constraints** with high-variance outcomes need careful handling (Lagrangian methods)
4. **Bandits are $\gamma = 0$ MDPs**—understanding this connection is foundational for Chapter 11

The Cold Start Problem (Ex 0.2) is the pedagogical highlight: We started with a hypothesis (local exploration is more efficient), discovered it was wrong, diagnosed why (cold start), and then validated when the intuition *does* hold (warm start). This is honest empiricism in action.

1.9 Running the Code

All solutions are in `scripts/ch00/lab_solutions.py`:

```bash
# Run all exercises
python scripts/ch00/lab_solutions.py --all

# Run specific exercise
python scripts/ch00/lab_solutions.py --exercise lab0.1
python scripts/ch00/lab_solutions.py --exercise 0.3

# Interactive menu
python scripts/ch00/lab_solutions.py
```

End of Lab Solutions