Contents

1 Chapter 6 — Exercises & Labs

This file contains exercises and labs for **Chapter 6: Discrete Template Bandits**.

Time budget: 90-120 minutes total **Difficulty calibration:** Graduate-level RL, assumes familiarity with probability, linear algebra, and Python

---

1.1 Theory Exercises (30 min total)

1.1.1 Exercise 6.1: Properties of Cosine Similarity (10 min)

Problem:

Consider semantic relevance $s_{\text{sem}}(\mathbf{q}, \mathbf{e}) = \frac{\mathbf{q} \cdot \mathbf{e}}{\|\mathbf{q}\|_2 \|\mathbf{e}\|_2}$ from [DEF-5.2] (Chapter 5 reference, used in template features).

Prove the following properties:

(a) $s_{\text{sem}}(\mathbf{q}, \mathbf{e}) \in [-1, 1]$ for all nonzero $\mathbf{q}, \mathbf{e} \in \mathbb{R}^d$

(b) $s_{\text{sem}}(\alpha\mathbf{q}, \beta\mathbf{e}) = \text{sign}(\alpha\beta) \cdot s_{\text{sem}}(\mathbf{q}, \mathbf{e})$ for all $\alpha, \beta \neq 0$

(c) If $\mathbf{e}_1, \mathbf{e}_2$ are orthogonal ($\mathbf{e}_1 \perp \mathbf{e}_2$), then $s_{\text{sem}}(\mathbf{q}, \mathbf{e}_1 + \mathbf{e}_2) \neq s_{\text{sem}}(\mathbf{q}, \mathbf{e}_1) + s_{\text{sem}}(\mathbf{q}, \mathbf{e}_2)$ in general

Hint: Use Cauchy-Schwarz inequality for (a). For (c), construct a counterexample.

Solution:

See `solutions/ex6_1.pdf` for detailed proof.

1.1.2 Exercise 6.2: Ridge Regression Closed Form (15 min)

Problem:

In [ALG-6.2] (LinUCB), we update weights via:

$$\hat{\theta}_a = A_a^{-1} b_a$$

where $A_a = \lambda I + \sum_{t:a_t=a} \phi_t \phi_t^\top$ and $b_a = \sum_{t:a_t=a} r_t \phi_t$.

(a) Prove this is equivalent to the ridge regression solution:

$$\hat{\theta}_a = \arg\min_{\theta} \left\{ \sum_{t:a_t=a} (r_t - \theta^\top \phi_t)^2 + \lambda \|\theta\|^2 \right\}$$

(b) Show that as $\lambda \to 0$, the solution converges to ordinary least squares (OLS):

$$\hat{\theta}_a^{\text{OLS}} = \left(\sum \phi_t \phi_t^\top \right)^{-1} \sum r_t \phi_t$$

(c) Explain why $\lambda > 0$ is critical for numerical stability when $\sum \phi_t \phi_t^\top$ is singular or ill-conditioned.

Hint: For (a), take the gradient of the objective, set to zero. For (c), discuss condition number.

Solution:

Proof of (a):

The objective is:

$$J(\theta) = \sum_{i=1}^{n} (r_i - \theta^\top \phi_i)^2 + \lambda \|\theta\|^2$$

Expanding:

$$J(\theta) = \sum_i (r_i^2 - 2r_i \theta^\top \phi_i + \theta^\top \phi_i \phi_i^\top \theta) + \lambda \theta^\top \theta$$

Taking the gradient with respect to θ:

$$\nabla_\theta J = \sum_i (-2r_i\phi_i + 2\phi_i\phi_i^\top\theta) + 2\lambda\theta$$

Setting $\nabla_\theta J = 0$:

$$\sum_i \phi_i\phi_i^\top\theta + \lambda\theta = \sum_i r_i\phi_i$$

Rearranging:

$$\left(\sum_i \phi_i\phi_i^\top + \lambda I\right)\theta = \sum_i r_i\phi_i$$

Thus:

$$\hat{\theta} = \left(\sum \phi_i\phi_i^\top + \lambda I\right)^{-1}\sum r_i\phi_i = A^{-1}b \quad \checkmark$$

Proof of (b):

As $\lambda \to 0$:

$$\hat{\theta}_a = \left(\sum \phi_t\phi_t^\top + \lambda I\right)^{-1}\sum r_t\phi_t \to \left(\sum \phi_t\phi_t^\top\right)^{-1}\sum r_t\phi_t = \hat{\theta}_a^{\text{OLS}}$$

assuming $\sum \phi_t\phi_t^\top$ is invertible.

Answer to (c):

When $\sum \phi_t\phi_t^\top$ is rank-deficient (e.g., fewer samples than features, or collinear features), OLS solution is undefined or numerically unstable (large condition number). Regularization λI adds λ to all eigenvalues, bounding the condition number:

$$\kappa(A) = \frac{\lambda_{\text{max}}(\sum \phi\phi^\top) + \lambda}{\lambda_{\text{min}}(\sum \phi\phi^\top) + \lambda} \leq \frac{\lambda_{\text{max}}}{\lambda}$$

This prevents numerical blow-up during matrix inversion.

1.1.3 Exercise 6.3: Thompson Sampling vs. LinUCB Posterior (5 min)

Problem:

Show that Thompson Sampling ([ALG-6.1]) and LinUCB ([ALG-6.2]) maintain **identical posterior mean $\hat{\theta}_a$** after observing the same data.

Specifically, verify that: - TS update: $\hat{\theta}_a = \Sigma_a(\Sigma_a^{-1}\hat{\theta}_a^{\text{old}} + \sigma^{-2}\phi r)$ - LinUCB update: $\hat{\theta}_a = A_a^{-1}b_a$ are equivalent when $\Sigma_a^{-1} = A_a$ and $\sigma^2 = 1$.

Solution:

Claim: Under $\Sigma_a^{-1} = A_a$ and $\sigma^2 = 1$, the posterior mean $\hat{\theta}_a$ is identical for Thompson Sampling ([ALG-6.1]) and LinUCB ([ALG-6.2]).

Thompson Sampling. After n observations $\{(\phi_i, r_i)\}_{i=1}^n$ for a fixed action a, the Bayesian linear regression update ([EQ-6.8]) gives

$$\Sigma_a^{-1} = \lambda I + \sum_{i=1}^n \phi_i \phi_i^\top,$$

and, with zero-mean prior,

$$\hat{\theta}_a = \Sigma_a \left(\sum_{i=1}^n r_i \phi_i \right).$$

LinUCB. With the same data, [ALG-6.2] maintains

$$A_a = \lambda I + \sum_{i=1}^n \phi_i \phi_i^\top, \qquad b_a = \sum_{i=1}^n r_i \phi_i,$$

and sets

$$\hat{\theta}_a = A_a^{-1} b_a.$$

Equivalence. Identifying $\Sigma_a^{-1} = A_a$ shows

$$\hat{\theta}_a^{\mathrm{TS}} = \Sigma_a \left(\sum_{i=1}^n r_i \phi_i \right) = A_a^{-1} b_a = \hat{\theta}_a^{\mathrm{LinUCB}}.$$

Thus both algorithms maintain the **same ridge regression estimate** for each action; the only difference lies in **action selection**: - Thompson Sampling samples $\tilde{\theta}_a \sim \mathcal{N}(\hat{\theta}_a, \Sigma_a)$ and selects $\arg\max_a \tilde{\theta}_a^\top \phi$ - LinUCB uses the deterministic UCB rule $\hat{\theta}_a^\top \phi + \alpha \sqrt{\phi^\top \Sigma_a \phi}$ and selects $\arg\max_a$

1.2 Implementation Exercises (40 min total)

1.2.1 Exercise 6.4: ε-Greedy Baseline (15 min)

Problem:

Implement an ε-greedy policy for template selection:

1. With probability ϵ: Select template uniformly at random
2. With probability $1 - \epsilon$: Select template with highest estimated mean reward

Use the same ridge regression updates as LinUCB, but replace UCB exploration with ε-greedy.

Specification:

```
class EpsilonGreedy:
    """Epsilon-greedy policy for contextual bandits.

    Args:
        templates: List of M boost templates
        feature_dim: Feature dimension d
        epsilon: Exploration rate in [0, 1]
        lambda_reg: Ridge regression regularization
```

```
        seed: Random seed
    """
    def __init__(self, templates, feature_dim, epsilon=0.1, lambda_reg=1.0, seed=42):
        # Initialize similar to LinUCB
        pass

    def select_action(self, features):
        """Select template using epsilon-greedy.

        With probability epsilon: random action
        With probability 1-epsilon: argmax_a theta_hat_a^T phi
        """
        # TODO: Implement
        pass

    def update(self, action, features, reward):
        """Ridge regression update (same as LinUCB)."""
        # TODO: Implement
        pass
```

Run on simulator for 50k episodes with $\epsilon \in \{0.05, 0.1, 0.2\}$. Compare cumulative regret to LinUCB.

Expected result: ε-greedy has **linear regret** $O(\epsilon T)$ (constant exploration never stops), while LinUCB has **sublinear regret** $O(\sqrt{T})$.

Solution:

See `solutions/ex6_4_epsilon_greedy.py` for implementation and plots.

1.2.2 Exercise 6.5: Cholesky-Based Thompson Sampling (20 min)

Problem:

The current TS implementation ([CODE-6.X]) samples from $\mathcal{N}(\hat{\theta}_a, \Sigma_a)$ by:

```
Sigma_a = np.linalg.inv(Sigma_inv[a])
theta_tilde = np.random.multivariate_normal(theta_hat[a], Sigma_a)
```

This requires matrix inversion **every episode** (expensive for large d).

Optimized approach: Maintain Cholesky factor L_a where $\Sigma_a^{-1} = L_a L_a^\top$. Sample via:

$$\tilde{\theta}_a = \hat{\theta}_a + L_a^{-T} z, \quad z \sim \mathcal{N}(0, I)$$

Implement this optimization:

1. Precompute $L_a = \text{cholesky}(\Sigma_a^{-1})$ after each update
2. Sample: Solve $L_a^\top v = z$ for v, then $\tilde{\theta}_a = \hat{\theta}_a + v$

Benchmark: Compare runtime for $d \in \{10, 50, 100, 500\}$ over 1000 episodes.

Expected speedup: 5-10× faster for $d \geq 100$.

Solution:

```
import numpy as np
from scipy.linalg import solve_triangular

class FastThompsonSampling:
    def __init__(self, M, d, lambda_reg=1.0):
        self.M, self.d = M, d
        self.theta_hat = np.zeros((M, d))
        self.Sigma_inv = np.array([lambda_reg * np.eye(d) for _ in range(M)])
        self.cholesky_factors = [np.linalg.cholesky(self.Sigma_inv[a]) for a in range(M)]

    def select_action(self, features):
        theta_samples = []
        for a in range(self.M):
            # Sample z ~ N(0, I)
            z = np.random.randn(self.d)

            # Solve L_a^T v = z for v
            v = solve_triangular(self.cholesky_factors[a].T, z, lower=False)

            # theta_tilde = theta_hat + v
            theta_tilde = self.theta_hat[a] + v
            theta_samples.append(theta_tilde)

        expected_rewards = [theta_samples[a] @ features for a in range(self.M)]
        return int(np.argmax(expected_rewards))

    def update(self, action, features, reward):
        a = action
        phi = features.reshape(-1, 1)

        # Update precision
        self.Sigma_inv[a] += phi @ phi.T

        # Update Cholesky factor
        self.cholesky_factors[a] = np.linalg.cholesky(self.Sigma_inv[a])

        # Update mean
        Sigma_a = np.linalg.inv(self.Sigma_inv[a])
        self.theta_hat[a] = Sigma_a @ (self.Sigma_inv[a] @ self.theta_hat[a] + phi.flatten() *
```

Benchmark code:

```
import time

for d in [10, 50, 100, 500]:
    # Naive TS
    policy_naive = LinearThompsonSampling(templates, d, ...)
```

```python
    start = time.time()
    for t in range(1000):
        features = np.random.randn(d)
        action = policy_naive.select_action(features)
        policy_naive.update(action, features, np.random.randn())
    time_naive = time.time() - start

    # Cholesky TS
    policy_fast = FastThompsonSampling(M=8, d=d, lambda_reg=1.0)
    start = time.time()
    for t in range(1000):
        features = np.random.randn(d)
        action = policy_fast.select_action(features)
        policy_fast.update(action, features, np.random.randn())
    time_fast = time.time() - start

    print(f"d={d:3d}: Naive {time_naive:.3f}s, Cholesky {time_fast:.3f}s, Speedup {time_naive/

# Expected output:
# d= 10: Naive 0.123s, Cholesky 0.098s, Speedup 1.3x
# d= 50: Naive 0.456s, Cholesky 0.089s, Speedup 5.1x
# d=100: Naive 1.234s, Cholesky 0.145s, Speedup 8.5x
# d=500: Naive 28.45s, Cholesky 2.341s, Speedup 12.2x
```

1.2.3 Exercise 6.6: Add Category Diversity Template (5 min)

Problem:

Extend the template library (Section 6.1.1) with a new template:

Template ID 8: Category Diversity

Boost products from **underrepresented categories** in the current result set to increase diversity.

Algorithm: 1. Count category frequencies in top-k results 2. Boost products from categories with count $< k/C$ where C is number of categories

Implementation:

```python
def create_diversity_template(catalog_stats, a_max=5.0):
    """Create category diversity boost template.

    Args:
        catalog_stats: Must include 'num_categories' (total categories C)
        a_max: Maximum boost

    Returns:
        template: BoostTemplate instance
    """
```

```python
C = catalog_stats['num_categories']

def diversity_boost_fn(p, result_set):
    """Compute diversity boost for product p given current result set.

    Args:
        p: Product dictionary with 'category' key
        result_set: List of products currently in top-k

    Returns:
        boost: Float in [0, a_max]
    """
    # Count categories in result_set
    category_counts = {}
    for prod in result_set:
        cat = prod['category']
        category_counts[cat] = category_counts.get(cat, 0) + 1

    # Expected uniform count
    k = len(result_set)
    expected_count = k / C

    # Product's category count
    p_cat = p['category']
    p_count = category_counts.get(p_cat, 0)

    # Boost if underrepresented
    if p_count < expected_count:
        return a_max * (1 - p_count / expected_count)
    else:
        return 0.0

return BoostTemplate(
    id=8,
    name="Category Diversity",
    description="Boost underrepresented categories",
    boost_fn=diversity_boost_fn
)
```

Task: Integrate this template into the library, run LinUCB with M=9 templates for 50k episodes. Report: 1. Selection frequency of diversity template 2. Catalog diversity metric: $H = -\sum_c p_c \log p_c$ where p_c is fraction of top-10 results from category c

Expected result: Diversity template selected in ~5-10% of episodes; diversity H increases by 0.2-0.5 nats.

1.3 Experimental Exercises (40 min total)

1.3.1 Lab 6.1: Reproducing the Simple-Feature Failure (20 min)

Objective: Reproduce the Section 6.5 experiment showing that contextual bandits with simple features underperform a strong static baseline.

Procedure:

1. Run the demo script in simple-feature mode:

```
python scripts/ch06/template_bandits_demo.py \
    --n-static 2000 \
    --n-bandit 20000 \
    --features simple
```

2. Record:
   - Best static template and its GMV (should be Premium with GMV ≈ 7.11)
   - LinUCB GMV (target ≈ 5.12)
   - Thompson Sampling GMV (target ≈ 6.18)

3. Compare LinUCB/TS to the best static template in terms of GMV and CM2.

4. Inspect the per-segment table printed by the script and identify at least two segments where bandits hurt GMV relative to the static winner.

Expected result: LinUCB ≈ -30 % GMV vs. static, TS ≈ -10 % GMV vs. static with clear per-segment losers. This is the Section 6.5 failure.

1.3.2 Lab 6.2a: Rich Features with Oracle Latents—Both Excel (15 min)

Objective: Re-run the experiment with rich features containing **true (oracle) user latents** (Section 6.7.4) and observe both algorithms performing excellently.

Procedure:

1. Run the demo with oracle latents:

```
python scripts/ch06/template_bandits_demo.py \
    --n-static 2000 \
    --n-bandit 20000 \
    --features rich \
    --rich-regularization blend \
    --prior-weight 50 \
    --lin-alpha 0.2 \
    --ts-sigma 0.5
```

2. Record:
   - Best static template and its GMV (Premium, GMV ≈ 7.11)
   - LinUCB GMV (target ≈ 9.42)
   - Thompson Sampling GMV (target ≈ 9.39)

3. Compute percentage lift vs. best static template for both algorithms.

Expected result: LinUCB $\approx$ +32% GMV vs. static, TS $\approx$ +32%. With clean oracle features, both algorithms perform excellently—nearly tied. This is Section 6.7.4.

1.3.3 Lab 6.2b: Rich Features with Estimated Latents—TS Wins (15 min)

Objective: Re-run the experiment with rich features containing **estimated (noisy) user latents** (Section 6.7.5) and observe Thompson Sampling's dominance.

Procedure:

1. Run the demo with estimated latents:

```
python scripts/ch06/template_bandits_demo.py \
    --n-static 2000 \
    --n-bandit 20000 \
    --features rich_est \
    --prior-weight 50 \
    --lin-alpha 0.2 \
    --ts-sigma 0.5
```

2. Record:
   - LinUCB GMV (target $\approx$ 7.52)
   - Thompson Sampling GMV (target $\approx$ 9.31)

3. Compute percentage lift vs. best static template for both algorithms.

Expected result: TS $\approx$ +31% GMV vs. static, LinUCB $\approx$ +6%. With noisy estimated features, Thompson Sampling's robust exploration wins. This is Section 6.7.5.

1.3.4 Lab 6.2c: Synthesis—The Algorithm Selection Principle (20 min)

Objective: Understand why the algorithm ranking reverses between oracle and estimated features, leading to the Algorithm Selection Principle (Section 6.7.6).

Procedure:

1. Create a 2x2 comparison table:

Features	LinUCB GMV	TS GMV	Winner	Margin
Oracle	~9.42	~9.39	LinUCB	+0.4 pts
Estimated	~7.52	~9.31	TS	+25 pts

2. Compute the "reversal magnitude": How many GMV points does the winner advantage change by?

3. **Key insight question:** Why does feature noise favor Thompson Sampling?

- Hint: Consider LinUCB's UCB bonus shrinkage vs. TS's perpetual posterior variance.

4. **Production question:** In a real e-commerce system, do you have oracle latents or estimated latents?
5. Write a 3-sentence recommendation for which algorithm to use in production.

1.4 Expected synthesis: Production systems have noisy estimated features, so Thompson Sampling should be your default. LinUCB is appropriate only when feature quality is exceptionally high (direct measurements, carefully validated estimates, or A/B test signals). This is Lesson 5.

1.4.1 Lab 6.3: Hyperparameter Sensitivity (20 min)

Objective: Understand how λ (regularization) and α (exploration) affect LinUCB performance.

Procedure:

1. Grid search over (λ, α):
   - $\lambda \in \{0.01, 0.1, 1.0, 10.0\}$
   - $\alpha \in \{0.1, 0.5, 1.0, 2.0, 5.0\}$
2. For each combination:
   - Train LinUCB for 50k episodes on `zoosim`
   - Record final average reward (last 10k episodes)
3. Plot heatmap: X-axis λ, Y-axis α, color = final reward
4. Identify optimal hyperparameters

Expected findings:

- λ too small (<0.1): Overfitting, unstable weights
- λ too large (>10): Underfitting, slow learning
- α too small (<0.5): Insufficient exploration, gets stuck
- α too large (>2): Excessive exploration, ignores rewards

Optimal region: $\lambda \in [0.5, 2.0]$, $\alpha \in [0.5, 1.5]$

Deliverable: Heatmap plot + 2-paragraph analysis of sensitivity

---

1.4.2 Lab 6.4: Visualization of Exploration Dynamics (15 min)

Objective: Visualize how bandits explore the template space over time.

Plots to create:

1. **Template selection heatmap**
   - X-axis: Episode (binned into 1000-episode windows)
   - Y-axis: Template ID (0-7)
   - Color: Selection frequency in window
   - Shows: How exploration decays, which templates dominate when
2. **Uncertainty evolution**

- X-axis: Episode
- Y-axis: trace(Σ_a) (total uncertainty) for each template
- Multiple lines (one per template)
- Shows: How uncertainty shrinks as data accumulates

3. **Regret decomposition**
   - X-axis: Episode
   - Y-axis: Cumulative regret
   - Stacked area chart: Regret contribution from each template
   - Shows: Which templates contribute most to regret (bad early selections)

Code template:

```python
import matplotlib.pyplot as plt
import seaborn as sns

# 1. Selection heatmap
window_size = 1000
num_windows = T // window_size
selection_matrix = np.zeros((M, num_windows))

...
```

1.4.3 Advanced Lab 6.A: From CPU Loops to GPU Batches (60–120 min)

This is an advanced, end-to-end lab that teaches you how and why to move from the canonical but slow Chapter 6 implementation under `scripts/ch06/` to the GPU-accelerated path under `scripts/ch06/optimization_gpu/`. It assumes you have completed at least Labs 6.1–6.3.

See the dedicated draft:

- `docs/book/drafts/ch06/ch06_advanced_gpu_lab.md`

for:

- Conceptual explanation of CPU vs GPU execution for template bandits
- A guided tour of `template_bandits_gpu.py`, `ch06_compute_arc_gpu.py`, and `run_bandit_matrix_gpu.py`
- Step-by-step tasks comparing CPU and GPU runs, exploring batch size and device choices, and extending diagnostics

This advanced lab is optional for first-time readers but **strongly recommended** if you plan to scale Chapter 6 experiments to many seeds, feature variants, or larger episode counts.

for w in range(num_windows): window_selections = selection_history[w*window_size:(w+1)*window_size] freqs = np.bincount(window_selections, minlength=M) / window_size selection_matrix[:, w] = freqs

plt.figure(figsize=(12, 6)) sns.heatmap(selection_matrix, cmap='viridis', cbar_kws={'label': 'Selection Frequency'}) plt.xlabel('Episode Window (x1000)') plt.ylabel('Template ID') plt.yticks(np.arange(M) + 0.5, [t.name for t in templates], rotation=0) plt.title('Template Selection Dynamics') plt.savefig('template_heatmap.png', dpi=150)

2 2. Uncertainty evolution

3 [Implement: Plot trace(Sigma_a) vs. episode for each template]

4 3. Regret decomposition

5 [Implement: Stacked area chart of per-template regret]

Deliverable: Three plots + interpretation paragraph for each

Lab 6.5: Multi-Seed Robustness (5 min)

Objective: Verify bandit performance is robust to random seed.

Procedure:

1. Run LinUCB with seeds $\{42, 123, 456, 789, 1011, 2022, 3033, 4044, 5055, 6066\}$
2. For each seed, record final average reward (last 10k episodes)
3. Compute mean $\pm$ std across seeds
4. Plot: Box plot of final rewards across seeds

Expected result:

- Mean final reward: ≈ 122 GMV
- Std: ≈ 2-3 GMV
- All seeds within $\pm 5\%$ of mean $\rightarrow$ robust

If variance is high (>5 GMV):
- Check: Are templates deterministic? (should be)
- Check: Is environment seed fixed? (should be independent per trial)
- Diagnosis: High variance suggests exploration randomness dominates $\rightarrow$ increase $T

Deliverable: Box plot + robustness assessment

Advanced Exercises (Optional, 30+ min)

Exercise 6.7: Hierarchical Templates (20 min)

Problem:

The flat template library has no structure. Design a **hierarchical template system**:

Level 1 (Meta-template): Select business objective
- Objective A: Maximize margin
- Objective B: Maximize volume (clicks/purchases)
- Objective C: Strategic goals

Level 2 (Sub-template): Given objective, select tactic
- If Objective A (margin): {High Margin, Premium, CM2}
- If Objective B (volume): {Popular, Discount, Budget}
- If Objective C (strategic): {Strategic, Category Diversity}

Bandit hierarchy:
1. Train meta-bandit over 3 objectives (context = user segment)
2. For each objective, train sub-bandit over tactics (context = query + product features)

Implementation sketch:

```python
class HierarchicalBandit:
    def __init__(self, meta_templates, sub_templates_dict, feature_dim):
        """
        Args:
            meta_templates: List of 3 objectives
            sub_templates_dict: Dict mapping objective_id -> list of sub-templates
            feature_dim: Context feature dimension
        """
        self.meta_bandit = LinUCB(meta_templates, feature_dim, ...)
        self.sub_bandits = {
            obj_id: LinUCB(templates, feature_dim, ...)
            for obj_id, templates in sub_templates_dict.items()
        }

    def select_action(self, features):
        """Two-stage selection."""
        # Stage 1: Select objective
        obj_id = self.meta_bandit.select_action(features)

        # Stage 2: Select tactic given objective
        tactic_id = self.sub_bandits[obj_id].select_action(features)

        return (obj_id, tactic_id)

    def update(self, action, features, reward):
        """Update both levels."""
        obj_id, tactic_id = action

        # Update sub-bandit (tactic level)
        self.sub_bandits[obj_id].update(tactic_id, features, reward)
```

```
        # Update meta-bandit (objective level)
        self.meta_bandit.update(obj_id, features, reward)
```

Task: Implement, run for 50k episodes, compare to flat LinUCB. Report: 1. Meta-level selection distribution (which objectives learned?) 2. Sub-level selection distribution per objective 3. Final GMV vs. flat LinUCB

Expected result: Hierarchical bandit achieves similar GMV with **faster convergence** (fewer parameters to learn per level) and **better interpretability** (business can understand objective → tactic mapping).

5.0.1 Exercise 6.8: Neural Linear Bandits (40 min) [Advanced, Optional]

!!! warning "Prerequisites for Exercise 6.8" This exercise requires: - Neural network implementation skills (PyTorch) - Understanding of representation learning (Chapter 12) - Pretraining on logged data (Chapter 13)

```
**If you haven't completed Chapter 12-13:** Skip this exercise or treat it as a reading exercis

**For advanced students:** This is a preview of techniques you'll use in deep RL chapters.
```

Problem:

Implement Neural Linear bandit as described in Appendix 6.A:

1. **Representation network:**
    - Input: Raw features (100-dim)
    - Architecture: [100 -> 64 -> 64 -> 20]
    - Activation: ReLU
2. **Pretraining:**
    - Collect 10k logged episodes using random template selection
    - Train network to predict reward: $r \approx \theta^\top f_\psi(x)$ where θ is linear head
    - Loss: MSE
    - Optimizer: Adam, lr=1e-3, 100 epochs
3. **Bandit training:**
    - Freeze representation f_ψ
    - Use LinUCB with features $\phi(x) = f_\psi(x)$

Comparison:

Run three conditions: - **Baseline:** LinUCB with hand-crafted features (Chapter 5) - **Neural Linear:** LinUCB with learned features f_ψ - **Oracle:** LinUCB with true optimal features (if known)

Metrics: - Sample efficiency: Episodes to reach 95% of final reward - Final GMV - Feature quality: R^2 of reward prediction on held-out set

Expected result:

Method	Sample Efficiency	Final GMV	Feature R^2
Hand-crafted	20k episodes	122.5	0.73
Neural Linear	15k episodes	124.2	0.81

Method	Sample Efficiency	Final GMV	Feature R^2
Oracle (unfair)	10k episodes	126.0	0.92

Conclusion: Neural Linear can improve over hand-crafted features **if sufficient pretraining data exists** (10k+ episodes). Otherwise, feature engineering is safer.

5.0.2 Exercise 6.9: Query-Conditional Templates (30 min)

Problem:

Current templates are **product-only** (don't depend on query). Extend to **query-conditional templates**:

Example: "Discount" template should: - Boost discounted products **more** for query `"deals"`, `"sale"` - Boost discounted products **less** for query `"premium dog food"`

Design:

Each template becomes:

$$t(p, q) = w_{\text{base}} \cdot f(p) + w_{\text{query}} \cdot g(q, p)$$

where: - $f(p)$: Product feature (margin, discount, etc.) - $g(q, p)$: Query-product interaction (e.g., cosine similarity of query to "discount" keywords) - $w_{\text{base}}, w_{\text{query}}$: Learned weights (bandit parameters)

Implementation:

1. Extend `BoostTemplate` to accept query as input:

   ```
   class QueryConditionalTemplate:
       def apply(self, products, query):
           return np.array([self.boost_fn(p, query) for p in products])
   ```

2. Augment features: $\phi(x) = [\phi_{\text{user}}(x), \phi_{\text{product}}(x), \phi_{\text{query}}(x)]$

3. Run LinUCB with augmented features

Evaluation:

Compare: - **Product-only templates** (baseline) - **Query-conditional templates** (proposed)

Metrics: - GMV by query type (navigational, informational, transactional) - Diversity of template selection per query type

Expected result: Query-conditional templates achieve **+3-5% GMV** on queries with strong intent signals (e.g., "cheap", "premium", "best") compared to product-only templates.

5.1 Solutions

Complete solutions with code, plots, and mathematical derivations are provided in:

- `solutions/theory/ex6_1_cosine_properties.pdf`
- `solutions/theory/ex6_2_ridge_regression.pdf`
- `solutions/theory/ex6_3_ts_linucb_equivalence.pdf`
- `solutions/implementation/ex6_4_epsilon_greedy.py`
- `solutions/implementation/ex6_5_cholesky_ts.py`
- `solutions/implementation/ex6_6_diversity_template.py`
- `solutions/labs/lab6_1_hyperparameters/` (heatmap + analysis)
- `solutions/labs/lab6_2_visualization/` (3 plots + interpretation)
- `solutions/labs/lab6_3_robustness/` (box plot + assessment)
- `solutions/advanced/ex6_7_hierarchical.py`
- `solutions/advanced/ex6_8_neural_linear/` (notebook + pretrained model)
- `solutions/advanced/ex6_9_query_conditional.py`

5.2 Time Allocation Summary

Category	Time (min)	Exercises
Theory	30	6.1 (10), 6.2 (15), 6.3 (5)
Implementation	40	6.4 (15), 6.5 (20), 6.6 (5)
Labs	80	Lab 6.1 (20), Lab 6.2 (20), Lab 6.3 (20), Lab 6.4 (15), Lab 6.5 (5)
Advanced (Optional)	90+	6.7 (20), 6.8 (40), 6.9 (30)
Total (Core)	150	
Total (with Advanced)	240+	

Recommended path:

- **Minimal (90 min):** Theory 6.1-6.3, Impl 6.4, Labs 6.1
- **Standard (110 min):** All core exercises + Labs 6.1-6.3
- **Deep dive (200 min):** Core + Advanced 6.7-6.9

Chapter 6 Exercises & Labs — First Draft — 2025