Contents

1 Chapter 6 Appendices: Advanced Topics and Extensions

Navigation: Back to Chapter 6 Main Text

This file contains advanced topics, theory-practice gap analysis, modern context, and production deployment guidance for Chapter 6. These sections are **optional** and intended for:

- **Practitioners** planning production deployment
- **Researchers** interested in theory-practice gaps and recent work (2020-2025)
- **Advanced students** exploring neural extensions and modern bandit applications

Prerequisites: Complete Chapter 6 core narrative (Sections 6.1–6.8) before reading these appendices.

1.1 Appendix 6.A: Neural Linear Bandits

Limitation of linear models:

The reward model $\mu(x, a) = \theta_a^\top \phi(x)$ assumes **hand-crafted features** $\phi(x)$ suffice. But what if: - Feature engineering is incomplete (missing interactions) - True reward function is highly nonlinear - We have high-dimensional raw inputs (images, text)

Solution: Learn feature representation with neural networks.

Neural Linear architecture:

$$\mu(x, a) = \theta_a^\top f_\psi(x) \tag{6.17}$$

{#EQ-6.17}

where: - $f_\psi : \mathcal{X} \to \mathbb{R}^d$ is a **neural network** with parameters ψ - $\theta_a \in \mathbb{R}^d$ is a **linear head** (bandit weights)

Why this design?

Option A: Fully neural bandit $\mu(x, a) = f_\theta(x, a)$ - **Pro:** Maximum expressiveness - **Con:** No closed-form posterior (requires MCMC or variational inference) - **Con:** Slow uncertainty quantification

Option B: Neural Linear (our choice) - **Pro:** Representation learning (neural net captures nonlinearity) - **Pro:** Fast uncertainty (linear head has Gaussian posterior) - **Pro:** Separates representation (ψ) from decision-making (θ_a) - **Con:** Still requires tuning representation network

Training procedure:

1. **Pretrain representation**: Use supervised learning on logged data to learn f_ψ
2. **Freeze representation**: Fix ψ, use $f_\psi(x)$ as features
3. **Run bandit**: Apply LinUCB/TS with features $\phi(x) = f_\psi(x)$

Alternatively, **joint training** (advanced): - Maintain replay buffer of (x, a, r) tuples - Periodically update ψ via gradient descent on prediction loss - Recompute bandit statistics with new features

Implementation sketch:

```
import torch
import torch.nn as nn

class NeuralFeatureExtractor(nn.Module):
    """Neural network for representation learning.

    Maps raw context x to learned features f_psi(x).
    """
    def __init__(self, input_dim: int, hidden_dim: int, output_dim: int):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Compute learned features f_psi(x).

        Args:
            x: Raw context, shape (batch, input_dim)

        Returns:
            features: Learned representation, shape (batch, output_dim)
        """
        return self.net(x)
```

```python
# Pretrain on logged data
feature_extractor = NeuralFeatureExtractor(input_dim=100, hidden_dim=64, output_dim=20)
optimizer = torch.optim.Adam(feature_extractor.parameters(), lr=1e-3)

for epoch in range(100):
    for x_batch, a_batch, r_batch in logged_data_loader:
        # Supervised learning: predict reward given (x, a)
        features = feature_extractor(x_batch)  # (batch, 20)
        # [Implement prediction head + loss]
        # [Backprop, optimizer.step()]
        pass

# Freeze and use with LinUCB
feature_extractor.eval()

def neural_linear_features(x):
    """Extract neural features for LinUCB."""
    with torch.no_grad():
        x_tensor = torch.tensor(x, dtype=torch.float32)
        return feature_extractor(x_tensor).numpy()

# Now use LinUCB with neural_linear_features instead of hand-crafted phi(x)
policy = LinUCB(templates, feature_dim=20, config=LinUCBConfig())

for t in range(T):
    obs, _ = env.reset()
    features = neural_linear_features(obs['raw_features'])  # Neural features!
    template_id = policy.select_action(features)
    # [Rest of training loop]
```

When to use Neural Linear:

- **Yes:** High-dimensional inputs (images, embeddings, raw text)
- **Yes:** Sufficient logged data for pretraining ($\geq$ 10k examples)
- **Yes:** Nonlinear reward structure not captured by hand-crafted features

When to avoid:

- **Avoid:** Limited data (<5k examples) — overfitting risk
- **Avoid:** Good hand-crafted features already exist — added complexity not justified
- **Avoid:** Interpretability critical — neural features are opaque

For our search ranking problem, **hand-crafted features suffice** (Chapter 5 engineering). Neural Linear is overkill unless we have raw query text or product images.

1.2 Appendix 6.B: Theory-Practice Gap Analysis

1.2.1 What Theory Guarantees

Theorem [THM-6.1] and [THM-6.2] prove:

Under assumptions: - (A1) **Linear mean rewards**: $\mu(x, a) = \theta_a^{*\top}\phi(x)$ - (A2) **Bounded features**: $\|\phi(x)\| \leq 1$ for all x - (A3) **Bounded parameters**: $\|\theta_a^*\| \leq S$ for all a - (A4) **Sub-Gaussian noise**: Reward noise has finite variance

Both LinUCB and Thompson Sampling achieve:

$$\text{Regret}(T) = O(d\sqrt{MT \log T})$$

1.2.2 What Our Implementation Does

Practice:

- **Violated:** (A1) Linearity: True reward likely nonlinear in features (saturation, interactions)
- **Satisfied:** (A2) Bounded features: We standardize features to $[-1, 1]$ range
- **Violated:** (A3) Bounded parameters: No explicit bound on $\|\theta_a\|$, rely on regularization
- **Satisfied:** (A4) Sub-Gaussian noise: Empirically observed (GMV variance finite)

1.2.3 Why It Works Anyway

1. Approximate linearity

While true $\mu(x, a)$ is nonlinear, our features $\phi(x)$ (Chapter 5) include: - Product-user interactions (margin $\times$ price sensitivity) - Category $\times$ query match - Popularity $\times$ discount interactions

These **approximate nonlinearity** via explicit interaction features. Linear model with good features $\approx$ nonlinear model with simple features.

2. Regularization bounds parameters

Ridge regression penalty $\lambda\|\theta\|^2$ implicitly enforces $\|\theta_a\| \lesssim \sqrt{\text{data scale}/\lambda}$. With $\lambda = 1.0$ and rewards ~100, we get $\|\theta_a\| \lesssim 10$ empirically.

3. Contextual structure

Even if model is misspecified, contextual bandits **still improve over static policies** because: - Features provide *some* signal about context-action affinity - Exploration discovers which templates work for which contexts - Worst case: Degrade to best static template (safe lower bound)

1.2.4 When It Fails

Failure Mode 1: Feature engineering is poor

If $\phi(x)$ doesn't capture context-action interactions, bandits converge to **random** or **best static** template.

Example: If features only include query length, product price (no interaction terms), linear model can't learn "cheap products for bargain hunters, premium for quality seekers".

Diagnostic: Selection frequencies converge to uniform or single template.

Fix: Add interaction features, polynomial features, or use Neural Linear (Appendix 6.A).

Failure Mode 2: Reward distribution shifts

If user behavior changes over time (seasonality, trend shifts), bandit posterior becomes **stale**.

Example: During holiday season, discount template optimal. Post-holiday, margin template optimal. Bandit trained on summer data performs poorly in winter.

Diagnostic: Increasing regret over time (non-convergent learning curves).

Fix: Discounted updates (give more weight to recent data), change-point detection (Chapter 15).

Failure Mode 3: Template library is incomplete

If optimal policy is *not representable* as any template (or convex combination), bandits hit **representation ceiling**.

Example: Optimal policy is "boost margin products for category=food, boost discount for category=toys". No single template captures this.

Diagnostic: Final performance plateaus below oracle.

Fix: Add category-specific templates, hierarchical templates, or move to continuous actions (Chapter 7).

1.2.5 Recent Work (2020-2025)

Contextual bandits remain active research:

1. Improved regret bounds

- [@agrawal:near_optimal:2017] shows TS achieves $O(d\sqrt{T})$ regret (no $\sqrt{M}$ factor) under realizability
- [@foster:practical:2020] proves instance-dependent bounds $O(\sqrt{dT/\Delta})$ where Δ is gap between best and second-best actions

2. Misspecification robustness

- [@kirschner:linear_bandits_misspec:2020] analyzes bandits under model misspecification: regret degrades gracefully as $O(\epsilon T^{2/3})$ where ϵ is approximation error
- Practical implication: **Even with nonlinear rewards, linear bandits work reasonably**

3. Neural bandits

- [@riquelme:deep_bayesian:2018] Neural Linear bandits for high-dimensional contexts
- [@zhou:neural_ts:2020] Thompson Sampling with neural network priors
- [@zhang:neural_ucb:2021] UCB for overparameterized neural networks with NTK analysis

4. Constrained bandits (relevant for Chapter 8)

- [@garcelon:conservative:2020] Safe exploration under CM2 floor constraints
- [@moradipari:primal_dual:2022] Primal-dual methods for contextual bandits with constraints

1.2.6 Open Problems

1. Why does Thompson Sampling work so well empirically?

Theory says $O(\sqrt{MT})$ but practice often sees $O(\sqrt{T})$ or better. Why? Posterior concentration is faster than worst-case analysis predicts (active research).

2. Optimal exploration for finite-sample regimes

Existing bounds are asymptotic ($T \to \infty$). For production with $T = 50k$, can we design algorithms with better **finite-sample** guarantees?

3. Multi-objective contextual bandits

Our reward is weighted sum (GMV, CM2, engagement). Can we learn **Pareto-optimal policies** without prespecifying weights? (Chapter 14 explores this.)

4. Adaptive feature selection

Which features matter for which actions? Can bandits **prune** features online to improve sample efficiency?

1.3 Appendix 6.C: Modern Context and Connections

1.3.1 Contextual Bandits in Production (2020-2025)

Industry deployments:

1. **Recommendation systems**
   - Netflix: Contextual bandits for homepage personalization [@gomez:netflix:2018]
   - YouTube: LinUCB for video recommendations at scale [@li:youtube_ucb:2010]
   - Spotify: Thompson Sampling for playlist generation [@mcinerney:spotify:2018]
2. **Search ranking**
   - Microsoft Bing: Contextual bandits for ad ranking [@agarwal:taming:2014]
   - Google: Bandits for query suggestion [@li:google:2016]
3. **E-commerce**
   - Amazon: Product recommendations and dynamic pricing [@kveton:amazon:2015]
   - Alibaba: Banner placement and promotional strategy [@tang:alibaba:2019]

Common themes:

- **Warm-start critical**: All use base ranker + learned boosts (our approach!)
- **Exploration budget**: Typically 5-15% of traffic for exploration
- **Guardrails**: Hard constraints on user experience metrics (click-through rate, diversity)

1.3.2 Connections to Deep RL

Contextual bandits as RL simplification:

- **Full RL**: $\pi(a|s, h)$ depends on state s and history h, optimizes $\mathbb{E}[\sum_t \gamma^t r_t]$
- **Contextual bandit**: $\pi(a|x)$ depends only on context x, optimizes $\mathbb{E}[r]$ (single-step)

When to use bandits vs. full RL:

Problem	Use Bandits If	Use Full RL If
Search ranking	Sessions are single-query	Multi-query sessions with retention effects (Chapter 11)
Recommendation	User watches one item then leaves	User browses playlist, next recommendation depends on history
Dynamic pricing	Price updated daily	Price affects inventory, long-term demand
Clinical trials	Patient outcomes independent	Treatment affects disease progression (multi-stage)

For search ranking (this chapter): Bandits suffice for **single-session optimization**. Chapter 11 extends to multi-session MDPs.

1.3.3 Bandits vs. Offline RL (Chapter 13)

Contextual bandits assume online interaction: - Bandit selects action → observes reward → updates policy - Requires live deployment (A/B testing, online learning)

Offline RL learns from logged data: - No online interaction during training - Must handle **distribution shift** (logged policy ≠ target policy)

Hybrid approach (used in practice): 1. **Phase 1 (Offline)**: Pretrain bandit on logged data from production system 2. **Phase 2 (Online)**: Deploy with exploration, continue learning 3. **Phase 3 (Refinement)**: Occasional offline updates from accumulated logs

Our implementation (Section 6.6) is **purely online**. Chapter 13 adds offline pretraining.

1.4 Appendix 6.D: Production Checklist

Before deploying template bandits in production, verify:

!!! tip "Production Checklist (Chapter 6)"

```
**Configuration Alignment:**
- [ ] Template library matches business objectives (`create_standard_templates` in `zoosim/poli
- [ ] Boost bounds $a_{\max} = 5.0$ aligned with `SimulatorConfig.action.a_max`
- [ ] Feature dimension $d$ matches `zoosim/ranking/features.py` output
- [ ] Regularization $\lambda \in [0.1, 10.0]$ tuned via cross-validation
- [ ] Exploration parameter $\alpha \in [0.5, 2.0]$ (LinUCB) or $\sigma \in [0.5, 2.0]$ (TS)

**Guardrails:**
- [ ] CM2 floor $\geq 60\%$ enforced (Chapter 8 adds hard constraints)
- [ ] Rank stability: $\Delta \text{rank}@k < 3$ positions per episode
- [ ] Fallback to best static template if bandit diverges (monitoring in Chapter 10)

**Reproducibility:**
- [ ] Seeds set consistently: `LinUCBConfig.seed`, `ThompsonSamplingConfig.seed`
- [ ] Logged data includes: (timestamp, context features, selected template, reward, metadata)
```

```
- [ ] Version templates library (commit hash) with logged data

**Monitoring (Chapter 10 details):**
- [ ] Track selection diversity (entropy of selection distribution)
- [ ] Log diagnostics: `theta_norms`, `uncertainty`, `selection_frequencies`
- [ ] Alert if cumulative regret grows linearly (indicates failure to learn)
- [ ] Dashboard: Learning curves, template performance by segment

**Testing:**
- [ ] Unit tests: `tests/ch06/test_templates.py`, `tests/ch06/test_linucb.py`
- [ ] Integration test: Full training loop on simulator with fixed seed
- [ ] Regression test: Final reward $\geq$ best static template - 5% (tolerance)
- [ ] Determinism test: LinUCB produces identical trajectory with same seed
```

Navigation: Back to Chapter 6 Main Text