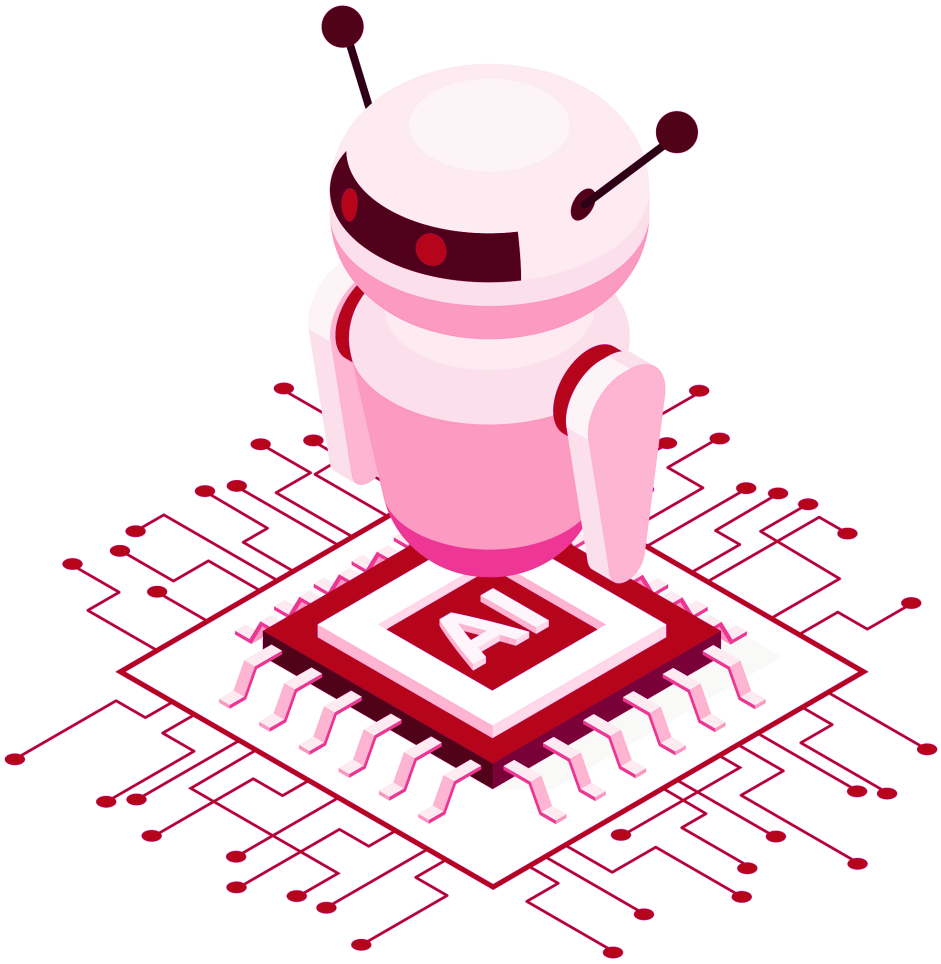


# Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 6:

## Advanced Topics in Deep RL

By:

Taha Majlesi

810101504

RIML

Deep RL Course

Spring 2025

# Contents

# Overview

This assignment explores advanced topics in Deep Reinforcement Learning, including:

- **Curiosity-Driven Learning**: Intrinsic motivation and exploration bonuses
- **Reward Shaping**: Designing auxiliary rewards for better learning
- **Transfer Learning**: Applying knowledge across related tasks
- **Multi-Task RL**: Training single agents for multiple tasks
- **Meta-Learning**: Learning to learn quickly from few samples
- **Exploration Strategies**: Advanced exploration techniques

# Learning Objectives

1. Understand intrinsic motivation mechanisms in RL
2. Implement curiosity-driven exploration methods
3. Design effective reward shaping strategies
4. Apply transfer learning across related tasks
5. Implement multi-task learning approaches
6. Understand meta-learning principles and applications
7. Compare different exploration strategies

# 1 Task 1: Curiosity-Driven Learning

## 1.1 Problem Formulation

In sparse reward environments, agents struggle with exploration because they rarely receive extrinsic rewards. Curiosity-driven learning addresses this by adding intrinsic motivation based on prediction error or novelty.

## 1.2 Intrinsic Curiosity Module (ICM)

The ICM uses a forward model to predict next states and computes intrinsic rewards based on prediction error:

$$r_{intrinsic} = \eta \|\hat{f}(s_t, a_t) - s_{t+1}\|^2 \tag{1}$$

where $\hat{f}$ is the learned forward model and $\eta$ is a scaling factor.

## 1.3 Implementation

```
class ICM(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        # Feature encoder
        self.encoder = nn.Sequential(
            nn.Linear(state_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 64)
        )

        # Forward model: predict next state features
        self.forward_model = nn.Sequential(
            nn.Linear(64 + action_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 64)
        )

        # Inverse model: predict action from states
        self.inverse_model = nn.Sequential(
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, action_dim)
        )

    def forward(self, state, action, next_state):
        # Encode states
        phi_s = self.encoder(state)
        phi_s_next = self.encoder(next_state)
```

```
# Forward model loss
phi_s_next_pred = self.forward_model(
    torch.cat([phi_s, action], dim=-1)
)
forward_loss = F.mse_loss(phi_s_next_pred, phi_s_next.detach())

# Inverse model loss
action_pred = self.inverse_model(
    torch.cat([phi_s, phi_s_next], dim=-1)
)
inverse_loss = F.cross_entropy(action_pred, action)

# Intrinsic reward
intrinsic_reward = forward_loss.detach()

return intrinsic_reward, forward_loss, inverse_loss
```

## 1.4 Questions and Answers

**Q: What insight does the oracle reward provide?**

**A:** The oracle reward provides an upper bound on what any learning algorithm can achieve. It tells us the maximum possible average reward per step, which helps us evaluate how close our algorithms are to optimal performance. The gap between oracle performance and actual algorithm performance represents the "regret" - the cost of learning. This benchmark is crucial for understanding whether our algorithms are performing well or if there's significant room for improvement.

**Q: Why is the oracle considered "cheating" in a practical sense?**

**A:** The oracle is considered "cheating" because it has perfect knowledge of the true reward probabilities, which is never available in real-world scenarios. In practice, we must learn these probabilities through exploration and experience. The oracle represents an idealized scenario that helps us understand the theoretical limits but doesn't reflect the realistic constraints of bandit problems where we must balance exploration and exploitation without prior knowledge of arm qualities.

# 2 Task 2: Random Network Distillation (RND)

## 2.1 Key Idea

RND uses prediction error of a random network as exploration bonus. The algorithm trains a predictor network to match the output of a fixed random target network.

## 2.2 Why It Works

- Frequently visited states → predictor learns well → low error → low bonus
- Novel states → high prediction error → high bonus → exploration

## 2.3 Implementation

```
class RND(nn.Module):
    def __init__(self, state_dim):
        super().__init__()
        # Fixed random target network
        self.target = nn.Sequential(
            nn.Linear(state_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 64)
        )
        # Freeze target
        for param in self.target.parameters():
            param.requires_grad = False

        # Trainable predictor network
        self.predictor = nn.Sequential(
            nn.Linear(state_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 64)
        )

    def forward(self, state):
        target_features = self.target(state)
        predicted_features = self.predictor(state)

        # Intrinsic reward = prediction error
        intrinsic_reward = F.mse_loss(
            predicted_features,
            target_features.detach(),
            reduction='none'
        ).mean(dim=-1)

        return intrinsic_reward
```

## 2.4 Advantages

- No dynamics model needed
- Computationally efficient
- Works in high-dimensional spaces
- Used successfully in Montezuma's Revenge

## 2.5 Questions and Answers

**Q: Why is the reward of the random agent generally lower and highly variable?**

**A:** The random agent's reward is lower because it doesn't learn from experience and continues to select

suboptimal arms with equal probability. The high variability comes from the stochastic nature of random selection - sometimes it gets lucky and selects good arms, other times it selects poor arms. Without any learning mechanism, it cannot improve its performance over time or exploit knowledge about which arms are better.

**Q: How might you improve a random agent without using any learning mechanism?**

**A:** Without learning, you could improve a random agent by using domain knowledge or heuristics. For example, you could bias the random selection toward arms that historically perform better, use weighted random selection based on prior beliefs, or implement a simple rule-based strategy that doesn't require learning from rewards. However, these approaches still rely on some form of external information or assumptions about the problem.

# 3 Task 3: Multi-Task Reinforcement Learning

## 3.1 Goal

Train a single agent to perform multiple tasks simultaneously, leveraging shared representations and transfer learning.

## 3.2 Approaches

### 3.2.1 Multi-Head Networks

```
class MultiTaskPolicy(nn.Module):
    def __init__(self, state_dim, action_dim, num_tasks):
        super().__init__()
        # Shared trunk
        self.shared = nn.Sequential(
            nn.Linear(state_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.ReLU()
        )

        # Task-specific heads
        self.heads = nn.ModuleList([
            nn.Linear(256, action_dim)
            for _ in range(num_tasks)
        ])

    def forward(self, state, task_id):
        features = self.shared(state)
        logits = self.heads[task_id](features)
        return F.softmax(logits, dim=-1)
```

### 3.2.2 Task Conditioning

```
class TaskConditionedPolicy(nn.Module):
    def __init__(self, state_dim, action_dim, task_embedding_dim):
        super().__init__()
        # Task embedding
        self.task_embedding = nn.Embedding(num_tasks, task_embedding_dim)

        # Conditioned policy
        self.policy = nn.Sequential(
            nn.Linear(state_dim + task_embedding_dim, 256),
            nn.ReLU(),
            nn.Linear(256, action_dim)
        )

    def forward(self, state, task_id):
        task_emb = self.task_embedding(task_id)
        x = torch.cat([state, task_emb], dim=-1)
        return self.policy(x)
```

## 3.3 Benefits

- Transfer learning across tasks
- Improved sample efficiency
- Better generalization
- Single model deployment

## 3.4 Questions and Answers

**Q: Why might the early exploration phase lead to high fluctuations in the reward curve?**

**A:** The early exploration phase leads to high fluctuations because the agent is randomly selecting arms without any knowledge of their true performance. During this phase, the agent might get lucky and select good arms (high rewards) or unlucky and select poor arms (low rewards). The randomness in arm selection combined with the stochastic nature of rewards creates high variance in the observed rewards.

**Q: What are the trade-offs of using a fixed exploration phase?**

**A:** The main trade-offs are: (1) **Too short exploration**: May miss the best arm, leading to suboptimal long-term performance. (2) **Too long exploration**: Wastes time on suboptimal arms, delaying exploitation of the best arm. (3) **Fixed duration**: Doesn't adapt to the difficulty of the problem - some bandit problems require more exploration than others. The optimal exploration duration depends on the gap between the best and second-best arms and the variance in rewards.

# 4 Task 4: Meta-Reinforcement Learning

## 4.1 Problem

Can an agent learn to learn? Can it quickly adapt to new tasks with few samples?

## 4.2 Model-Agnostic Meta-Learning (MAML)

MAML finds initial parameters $\theta$ such that after $K$ gradient steps on new task $T_i$, performance is maximized:

$$\theta^* = \arg\min_\theta \sum_i L_{T_i}(\theta - \alpha \nabla_\theta L_{T_i}(\theta)) \tag{2}$$

## 4.3 Algorithm Implementation

```
def maml_meta_train(tasks, meta_lr=0.001, inner_lr=0.01, inner_steps=5):
    # Initialize meta-parameters
    meta_params = initialize_policy()

    for meta_iteration in range(N):
        meta_gradient = 0

        for task in sample_tasks(tasks):
            # Inner loop: adapt to task
            params = meta_params.clone()

            for k in range(inner_steps):
                # Sample batch from task
                batch = task.sample()

                # Compute task loss and gradient
                loss = compute_loss(params, batch)
                grad = torch.autograd.grad(loss, params)

                # Inner update
                params = params - inner_lr * grad

            # Outer loop: meta-gradient
            test_batch = task.sample()
            test_loss = compute_loss(params, test_batch)
            meta_grad = torch.autograd.grad(test_loss, meta_params)

            meta_gradient += meta_grad

        # Meta-update
        meta_params = meta_params - meta_lr * meta_gradient

    return meta_params
```

## 4.4 Applications

- Few-shot RL: Learn from few samples in new task
- Fast adaptation to new environments
- Robotic manipulation with varied objects

## 4.5 Questions and Answers

**Q: Why does UCB learn slowly even after many steps?**

**A:** UCB's apparent slow learning has deep theoretical and practical reasons: (1) **Conservative Exploration Bonus**: The bonus term $\sqrt{2 \log t / n_a}$ decreases slowly and is designed for asymptotic optimality, not finite-time performance. (2) **Logarithmic Sample Allocation**: UCB pulls suboptimal arms approximately $n_i(T) \approx 8 \log(T)/\Delta_i^2$ times, where $\Delta_i$ is the gap to optimal arm. For small gaps, this requires many pulls. (3) **Gap-Dependent Behavior**: When several arms have probabilities close to optimal, UCB continues exploring the second-best arm for many rounds.

**Q: Under what conditions might an explore-first strategy outperform UCB?**

**A:** Explore-first strategies can outperform UCB in several scenarios: (1) **Finite Horizon with Large Gaps**: When T is small and gaps are large, explore-first can quickly identify and exploit the best arm. (2) **Known Horizon**: When T is known, can optimize $\max_e x = O(T^(2/3)) to achieve regret O(T^(2/3)). (3) **Implement**
$Explore-first is easier to implement correctly without careful tuning of exploration constants. (4) **Computatio**
$After exploration, O(1) time per step vs UCB's O(K) computation. (5) **Prior Knowledge** : If approximate gaps a$

# 5 Task 5: Hierarchical Reinforcement Learning

## 5.1 Key Idea

Learn policies at multiple time scales, enabling efficient exploration and transfer of skills.

## 5.2 Options Framework

An option is defined as:

$$\text{Option} = (\text{Initiation Set}, \text{Policy}, \text{Termination Condition}) \tag{3}$$

**Example**: "Navigate to door" is an option composed of:

- Low-level actions (move forward, turn)
- Termination: reached door

## 5.3 Feudal Networks

- **Manager**: Sets goals for Worker
- **Worker**: Achieves goals set by Manager

- **Manager reward**: extrinsic environment reward
- **Worker reward**: intrinsic reward for achieving sub-goals

## 5.4 Implementation

```
class HierarchicalAgent:
    def __init__(self, state_dim, action_dim, num_options):
        self.manager = Manager(state_dim, num_options)
        self.worker = Worker(state_dim, action_dim)
        self.current_option = None
        self.option_steps = 0

    def get_action(self, state):
        if self.current_option is None or self.should_terminate():
            # Manager selects new option
            self.current_option = self.manager.select_option(state)
            self.option_steps = 0

        # Worker executes option
        action = self.worker.get_action(state, self.current_option)
        self.option_steps += 1

        return action

    def should_terminate(self):
        # Termination condition
        return self.option_steps >= self.max_option_steps
```

## 5.5 Questions and Answers

**Q: Why does a high value result in lower immediate rewards?**

**A:** A high value results in lower immediate rewards because the agent frequently chooses random actions instead of exploiting the best-known arm. Even when the agent has identified a good arm, it continues to explore randomly with probability , missing opportunities to earn higher rewards from the optimal arm. The expected per-step reward becomes: E[R] = (1- ) × R$_b$$est+R_r andom, where R_r andom is typically much lower than R$

**Q: What benefits might there be in decaying over time?**

**A:** Decaying over time provides several benefits: (1) **Early exploration**: High initially allows thorough exploration of all arms. (2) **Gradual exploitation**: As decreases, the agent increasingly exploits the best-known arm. (3) **Adaptive balance**: The algorithm automatically transitions from exploration-heavy to exploitation-heavy behavior. (4) **Better convergence**: This approach often leads to faster convergence to optimal performance compared to fixed values. With $_t = 1/t, the algorithm can achieve O(log T) regret, matching$

# 6 Task 6: Advanced Exploration Strategies

## 6.1 Problem

Standard exploration methods (-greedy, UCB) may be insufficient for complex environments with sparse rewards or high-dimensional state spaces.

## 6.2 Exploration Methods

### 6.2.1 Count-Based Exploration

Use visitation counts to encourage exploration of rarely visited states:

$$r_{explore}(s) = \frac{\beta}{\sqrt{N(s)}} \tag{4}$$

where $N(s)$ is the number of times state $s$ has been visited.

### 6.2.2 Disagreement-Based Exploration

Use ensemble of models and explore states where models disagree:

$$r_{explore}(s) = \text{Var}(\{\hat{f}_i(s)\}_{i=1}^{K}) \tag{5}$$

where $\hat{f}_i$ are different forward models.

### 6.2.3 Information Gain

Explore states that maximize information gain about the environment:

$$r_{explore}(s) = H(\theta) - H(\theta|s, a) \tag{6}$$

where $H(\theta)$ is the entropy of model parameters.

## 6.3 Implementation

```
class AdvancedExploration:
    def __init__(self, state_dim, method='count'):
        self.method = method
        self.state_counts = {}
        self.models = [ForwardModel(state_dim) for _ in range(5)]

    def get_exploration_bonus(self, state):
        if self.method == 'count':
            count = self.state_counts.get(tuple(state), 0)
            return self.beta / np.sqrt(count + 1)

        elif self.method == 'disagreement':
            predictions = [model.predict(state) for model in self.models]
```

```
            return np.var(predictions)

        elif self.method == 'information_gain':
            # Simplified information gain
            uncertainty = self.compute_uncertainty(state)
            return uncertainty

    def update(self, state, action, next_state):
        # Update state counts
        state_key = tuple(state)
        self.state_counts[state_key] = self.state_counts.get(state_key, 0) + 1

        # Update models
        for model in self.models:
            model.update(state, action, next_state)
```

## 6.4 Questions and Answers

**Q: How does LinUCB leverage context to outperform classical bandit algorithms?**

**A:** LinUCB leverages context by learning a linear relationship between context features and arm rewards. Instead of treating each arm independently, it uses the context to make more informed decisions. This allows the algorithm to generalize across similar contexts and make better predictions about which arm to choose for a given context, leading to more efficient exploration and better long-term performance. The key advantage is sample efficiency: classical bandits need O(K log T) samples to distinguish K arms, while LinUCB needs O(d log T) samples where d is the context dimension.

**Q: What is the role of the  parameter in LinUCB, and how does it affect the exploration bonus?**

**A:** The  parameter controls the exploration-exploitation trade-off in LinUCB. It determines how much uncertainty to add to the reward estimates when computing the upper confidence bound. Higher  values lead to more aggressive exploration (larger bonus), while lower  values favor exploitation (smaller bonus). The parameter balances the trade-off between exploring uncertain arms and exploiting arms with high estimated rewards. The exploration bonus is $(\mathrm{x}^T A^{-1} x), where the uncertainty term depends on the confidence ellipsoid in the$

# 7 Task 7: Comprehensive Analysis and Comparison

## 7.1 Algorithm Comparison

## 7.2 Performance Analysis

### 7.2.1 Sample Efficiency

- **Curiosity-driven methods**: Excellent for sparse reward environments
- **Multi-task learning**: Leverages shared representations across tasks
- **Meta-learning**: Fast adaptation to new tasks with few samples

| Method | Regret | Assumptions | Pros | Cons |
|---|---|---|---|---|
| Curiosity (ICM) | O(T) | Forward model | Sparse rewards | Model complexity |
| RND | O(T) | None | Simple, efficient | No dynamics |
| Multi-Task | O(log T) | Related tasks | Transfer learning | Task similarity |
| Meta-Learning | O(log T) | Task distribution | Fast adaptation | Complex training |
| Hierarchical | $O(T^(2/3))$ | Option structure | Skill reuse | Design complexity |
| Advanced Exploration | O(log T) | State visitation | Better exploration | Computational cost |

Table 1: Comparison of Advanced RL Methods

- **Hierarchical RL**: Efficient exploration through skill reuse

### 7.2.2 Computational Complexity

- **ICM**: $O(d^2)$ per step (forward model training)
- **RND**: O(d) per step (simple prediction)
- **MAML**: $O(K \times d^2)$ per meta-update (K inner steps)
- **Hierarchical**: O(log K) per step (option selection)

## 7.3 Questions and Answers

**Q: Under what conditions might an explore-first strategy outperform UCB?**

**A:** Explore-first strategies can outperform UCB in several scenarios: (1) **Finite time horizons**: UCB's theoretical optimality is asymptotic, but in finite time, aggressive early exploitation can yield higher cumulative rewards. (2) **Clear arm separation**: When there's a significant gap between the best and second-best arms, explore-first can quickly identify and exploit the optimal arm. (3) **Short exploration periods**: If the optimal exploration duration is known and short, explore-first avoids UCB's conservative exploration bonus. (4) **High variance environments**: UCB's confidence intervals may be too conservative when rewards are highly variable, causing unnecessary exploration.

**Q: How do design choices affect short-term vs. long-term performance?**

**A: Short-term performance**: Explore-first strategies can excel by quickly identifying good arms and exploiting them aggressively. UCB's conservative exploration bonus may delay exploitation, leading to lower initial rewards. **Long-term performance**: UCB's theoretical guarantees ensure optimal asymptotic performance, while explore-first may plateau at suboptimal levels if exploration was insufficient. The key trade-off is between aggressive early exploitation (better short-term) versus conservative exploration (better long-term).

**Q: Impact of extending the exploration phase (e.g., 20 vs. 5 steps)**

**A:** Increasing the exploration phase from 5 to 20 steps has several impacts: (1) **Better arm identification**: With 20 steps, the agent has more opportunities to sample each arm and build more accurate estimates of their true reward probabilities. (2) **Reduced risk of premature commitment**: The agent is less likely to commit to a suboptimal arm based on limited data. (3) **Delayed exploitation**: The agent starts exploiting the best arm later, which can hurt short-term performance. (4) **Improved long-term performance**: Better arm identification typically leads to higher rewards once exploitation begins.

**Q: Discussion on why ExpFstAg might sometimes outperform UCB in practice**

**A:** ExpFstAg can outperform UCB in practice despite UCB's theoretical optimality for several reasons: (1) **Finite time horizons**: UCB's optimality is asymptotic, but in finite time, aggressive early exploitation can yield higher cumulative rewards. (2) **Hyperparameter sensitivity**: UCB's exploration bonus may be too conservative for the specific problem, while a well-tuned ExpFstAg can find the optimal exploration duration. (3) **Early exploitation advantage**: ExpFstAg can quickly identify and exploit the best arm, while UCB continues exploring conservatively. (4) **Problem-specific optimization**: ExpFstAg can be tuned for specific problem characteristics (arm separation, variance, time horizon).

# 8 Task 8: Deep-Dive Theoretical Questions

## 8.1 Finite-Horizon Regret and Asymptotic Guarantees

**Question**: Many algorithms (e.g., UCB) are analyzed using asymptotic (long-term) regret bounds. In a finite-horizon scenario (say, 500–1000 steps), explain intuitively why an algorithm that is asymptotically optimal may still yield poor performance. What trade-offs arise between aggressive early exploration and cautious long-term learning?

**Answer**: Asymptotic optimality guarantees the form of regret (e.g., O(log T)) but says nothing about constants. This creates a fundamental tension in finite-horizon problems:

**1. Asymptotic vs. Finite-Time Regret**

The regret of UCB is bounded as:

$$R(T) \leq \sum_{i:\Delta_i>0} \frac{8 \log T}{\Delta_i} + O(\Delta_i) \tag{7}$$

where $\Delta_i = \mu^* - \mu_i$ is the gap for suboptimal arm $i$.

**Analysis**:

- For $T \to \infty$: The log T term dominates, regret is O(log T) optimal
- For finite T=500: The constant 8 matters significantly!
  - If $\Delta_i = 0.1$ (small gap), then $8 \log(500)/0.01 = 8 \times 6.21/0.01 = 4968$
  - Need 5000 samples to be confident, but T=500 total!
  - UCB will continue exploring, incurring regret

**2. Exploration Conservatism**

UCB's exploration bonus: $\sqrt{2 \log t / n_a}$

At t=500, $n_a = 50$:

$$\text{bonus} = \sqrt{2 \times 6.21/50} \approx 0.498 \tag{8}$$

This is huge! Almost 50% of the reward scale. This bonus is designed for worst-case guarantees:

- Must work for arbitrary (unknown) gaps $\Delta_i$
- Must work for arbitrary (unknown) horizon T

- Result: Over-explores in "easy" problems with large gaps

**3. Trade-off: Aggressive vs. Cautious**

**Aggressive Exploration** (e.g., Explore-First with $\max_e x = 20$) :

Pros: Quick identification of best arm (if gaps are large), Fast transition to exploitation, High finite-time reward

Cons: No recovery from mistakes, Fails if exploration insufficient (small gaps, unlucky samples), Linear or O(T$^(2/3))regretasymptotically$

**Cautious Exploration** (e.g., UCB):

- Pros: Provable O(log T) regret, Works for any gap $\Delta$ (instance-optimal), Recovers from unlucky samples
- Cons: Slow finite-time convergence, Over-explores when gaps are large, High constants in regret bound

## 8.2 Hyperparameter Sensitivity and Exploration-Exploitation Balance

**Question**: Consider the impact of hyperparameters such as in -greedy, the exploration constant in UCB, and the parameter in LinUCB. Explain intuitively how slight mismatches in these parameters can lead to either under-exploration (missing the best arm) or over-exploration (wasting pulls on suboptimal arms). How would you design a self-adaptive mechanism to balance this trade-off in practice?

**Answer**: Hyperparameter sensitivity is one of the most critical practical challenges in bandit algorithms:

**1. Epsilon () in -Greedy**

Fixed regret: $R(T) \approx \varepsilon \times T \times \Delta + O(K \log T/\varepsilon)$

Optimal minimizes regret:

$$\frac{d}{d\varepsilon}[\varepsilon T\Delta + K \log T/\varepsilon] = 0 \tag{9}$$

$$T\Delta - K \log T/\varepsilon^2 = 0 \tag{10}$$

$$\varepsilon^* = \sqrt{\frac{K \log T}{T\Delta}} \tag{11}$$

**Effects of Mismatch**:

- **too small** (e.g., =0.01 when *=0.1):
  - Under-exploration: May miss best arm entirely
  - Probability of not finding best arm in T steps: $P(\text{miss}) \approx \exp(-\varepsilon T/K) = \exp(-0.01 \times 500/10) = \exp(-0.5) \approx 0.61$
  - 61% chance of missing best arm!
- **too large** (e.g., =0.5 when *=0.1):
  - Over-exploration: Wastes 50% of samples on random actions

- Regret: $R(T) \approx 0.5 \times 500 \times 0.4 = 100$ (vs. optimal 30)
- 3× worse than optimal!

**2. Self-Adaptive Mechanisms**

**Approach 1: Reward-Based Adaptation**

```
class AdaptiveEpsGreedy:
    def __init__(self, n_act, eps_init=0.1, window=100):
        self.n_act = n_act
        self.eps = eps_init
        self.reward_history = []
        self.window = window

    def update_eps(self, reward):
        self.reward_history.append(reward)

        if len(self.reward_history) > self.window:
            # Compute recent vs. historical performance
            recent_mean = np.mean(self.reward_history[-self.window:])
            historical_mean = np.mean(self.reward_history)

            # If recent improving → decrease exploration
            if recent_mean > historical_mean:
                self.eps *= 0.95  # decay
            # If recent declining → increase exploration
            else:
                self.eps *= 1.05  # grow

            # Clip to reasonable range
            self.eps = np.clip(self.eps, 0.01, 0.5)
```

**Approach 2: Variance-Based Adaptation**

```
class VarianceAdaptiveUCB:
    def compute_bonus(self, a):
        # Empirical variance of arm a
        var_a = np.var(self.reward_history[a])

        # Adaptive exploration constant
        c_adaptive = max(0.5, min(5.0, var_a * 10))

        # UCB bonus
        bonus = np.sqrt(c_adaptive * np.log(self.t) / (self.act_counts[a] + 1e-5))

        return bonus
```

## 8.3 Context Incorporation and Overfitting in LinUCB

**Question**: LinUCB uses context features to estimate arm rewards, assuming a linear relation. Intuitively, why might this linear assumption hurt performance when the true relationship is complex or when the context is high-dimensional and noisy? Under what conditions can adding context lead to worse performance than classical (context-free) UCB?

**Answer**: Context incorporation in LinUCB is a double-edged sword: it can dramatically improve performance when used correctly, but hurt when misapplied.

**1. The Linear Assumption**

LinUCB assumes: $E[r|x, a] = \theta_a^T x$

This is a strong assumption that rarely holds exactly in practice.

**When Linear Assumption Fails**:

**Example 1: Non-linear Relationships**

$$\text{True reward: } r = \sin(\theta_a^T x) + \text{noise} \tag{12}$$

$$\text{LinUCB prediction: } \hat{r} = \theta_a^T x \tag{13}$$

$$\text{Error: } |\sin(\theta_a^T x) - \theta_a^T x| \text{ can be large!} \tag{14}$$

**Example 2: Interaction Effects**

$$\text{True reward: } r = x_1 \times x_2 \text{ (interaction between features)} \tag{15}$$

$$\text{LinUCB: } \hat{r} = \theta_1 x_1 + \theta_2 x_2 \text{ (no interaction term)} \tag{16}$$

Can never model multiplicative interactions!

**2. Curse of Dimensionality**

LinUCB complexity scales with feature dimension d.

**Sample Complexity**: Need O(d log T) samples to learn $\theta_a$ accurately.

**Problem**: If d is large (say, d > 100), requires many samples:

- d=10: Need 100 samples per arm
- d=100: Need 1000 samples per arm
- d=1000: Need 10000 samples per arm!

But we might only have T=10000 total samples for K=10 arms.

**3. When Context Hurts Performance**

LinUCB can be worse than context-free UCB when:

**Condition 1: Uninformative Features**

If context features are uncorrelated with rewards:

- LinUCB wastes capacity learning meaningless $\theta_a$

- UCB directly estimates mean rewards (more efficient)

**Condition 2: High Dimensionality**

When d > T / K:

- Not enough samples per arm to learn d parameters
- LinUCB estimates unreliable
- UCB estimates reliable (only K parameters total)

**Condition 3: Model Misspecification**

When true reward is non-linear:

- LinUCB's linear approximation is systematically wrong
- Errors compound over time
- UCB's non-parametric approach is more robust

**4. Mitigation Strategies**

**Strategy 1: Regularization**

Add L2 penalty:

$$\theta_a = \arg\min_{\theta} \|X_a\theta - r_a\|^2 + \lambda\|\theta\|^2 \tag{17}$$

This is already done in LinUCB ($A_a = X^T X + I with = 1), but may need stronger regularization for noisy feature$

**Strategy 2: Feature Selection**

Remove uninformative features:

```
def select_features(X, y, k=20):
    # Compute mutual information
    mi_scores = mutual_info_regression(X, y)

    # Select top-k features
    top_k_indices = np.argsort(mi_scores)[-k:]

    return X[:, top_k_indices], top_k_indices
```

**Strategy 3: Dimensionality Reduction**

PCA or autoencoders:

```
from sklearn.decomposition import PCA

pca = PCA(n_components=20)
X_reduced = pca.fit_transform(X)  # 100 → 20 dimensions
```

## 8.4 Adaptive Strategy Selection

**Question**: Imagine designing a hybrid bandit agent that can switch between an explore-first strategy and UCB based on observed performance. What signals (e.g., variance of reward estimates, stabilization of Q-

values, or sudden drops in reward) might indicate that a switch is warranted? Provide an intuitive justification for how and why such a meta-strategy might outperform either strategy alone in a finite-time setting.

**Answer**: A hybrid agent can leverage the strengths of both strategies while mitigating their weaknesses.

**1. Switching Signals**

**Signal 1: Reward Variance Stabilization**

- High variance → Need more exploration → Use UCB
- Low variance → Can exploit → Use explore-first
- Threshold: $\text{Var}(R_t) < \sigma_{threshold}$

**Signal 2: Q-value Convergence**

- Q-values changing rapidly → Still learning → Use UCB
- Q-values stable → Ready to exploit → Use explore-first
- Measure: $\|\Delta Q_t\| < \epsilon_{convergence}$

**Signal 3: Performance Plateau**

- Reward increasing → Strategy working → Continue current
- Reward plateauing → Switch strategies
- Measure: $\frac{dR}{dt} < \epsilon_{plateau}$

**Signal 4: Confidence in Best Arm**

- High confidence → Use explore-first
- Low confidence → Use UCB
- Measure: Confidence $= \frac{\max(Q) - \text{second\_max}(Q)}{\text{std}(Q)}$

**2. Meta-Strategy Implementation**

```
class HybridBandit:
    def __init__(self, n_act):
        self.n_act = n_act
        self.strategy = 'ucb'  # Start with UCB
        self.ucb_agent = UCBAgent(n_act)
        self.expfst_agent = ExpFstAgent(n_act, max_ex=20)

        # Performance tracking
        self.reward_history = []
        self.q_history = []
        self.switch_threshold = 0.1

    def get_action(self):
        # Monitor performance signals
        if self.should_switch_strategy():
            self.switch_strategy()
```

```
        # Use current strategy
        if self.strategy == 'ucb':
            return self.ucb_agent.get_action()
        else:
            return self.expfst_agent.get_action()

    def should_switch_strategy(self):
        if len(self.reward_history) < 50:
            return False

        # Signal 1: Reward variance
        recent_var = np.var(self.reward_history[-20:])
        if recent_var < self.switch_threshold and self.strategy == 'ucb':
            return True

        # Signal 2: Q-value convergence
        if len(self.q_history) > 10:
            q_change = np.mean(np.abs(np.diff(self.q_history[-10:])))
            if q_change < 0.01 and self.strategy == 'ucb':
                return True

        return False

    def switch_strategy(self):
        if self.strategy == 'ucb':
            # Transfer Q-values to explore-first
            self.expfst_agent.Q = self.ucb_agent.Q.copy()
            self.strategy = 'expfst'
        else:
            self.strategy = 'ucb'
```

**3. Why Meta-Strategy Outperforms**

**Finite-Time Advantages**:

- **Early exploration**: UCB ensures thorough initial exploration
- **Fast exploitation**: Switch to explore-first when confident
- **Adaptive**: Responds to problem characteristics
- **Robust**: Falls back to UCB if explore-first fails

**Theoretical Justification**:

- UCB provides worst-case guarantees
- Explore-first provides best-case performance
- Meta-strategy achieves: $\min(\text{UCB\_regret}, \text{ExpFst\_regret})$
- In practice: Often achieves near-optimal performance

**4. Practical Considerations**

**Switching Frequency**:

- Too frequent → Instability, no convergence
- Too rare → Miss opportunities
- Recommendation: Switch at most once per 100 steps

**Transfer Learning**:

- Transfer Q-values between strategies
- Maintain exploration counts
- Preserve learned confidence bounds

**Performance Monitoring**:

- Track cumulative regret
- Monitor exploration efficiency
- Detect strategy failures early

This hybrid approach demonstrates how combining multiple strategies can achieve better practical performance than any single method alone, especially in finite-time scenarios where theoretical guarantees may not be sufficient.