Case Study: Adaptive Content Recommendation with Policy Gradient Methods

NovaMind AI -- Personalizing Learning Paths in Real-Time

Section 1: Industry Context and Business Problem

Industry: EdTech -- Adaptive Learning Platforms

The global edtech market reached \$340 billion in 2025, with adaptive learning platforms representing the fastest-growing segment. Unlike static content delivery, adaptive systems observe learner behavior in real time and adjust the sequence and difficulty of presented material to maximize learning outcomes.

Company Profile: NovaMind AI

NovaMind AI is a Series B edtech startup (\$45M raised) headquartered in San Francisco. They operate an adaptive learning platform serving 2.3 million users across K-12, university, and corporate training verticals. Their platform delivers microlearning modules -- short, focused lessons of 3-15 minutes each.

Revenue model: \$18/month per individual user, \$12/month per enterprise seat (minimum 500 seats).

Current annual recurring revenue (ARR): \$31M with 38% year-over-year growth.

The Business Challenge

NovaMind's current content recommendation system uses a rule-based approach: a fixed decision tree that assigns content based on pre-test scores and completion rates. This system has three critical problems:

1. **One-size-fits-all sequencing.** A learner who scores 70% on a pre-test gets the same content sequence as every other learner with a 70% score, regardless of their learning pace, preferred content type, or knowledge gaps.

2. **Delayed adaptation.** The system only re-evaluates content assignments after a learner completes a full module (10-15 minutes). Learners who are struggling receive no intervention during the module itself.

3. **No exploration.** The system never tries novel content sequences. It always recommends the "safe" path, missing opportunities to discover that some learners benefit from advanced content earlier or from revisiting fundamentals in a different order.

Business impact: - Learner completion rates have stagnated at 34% - Average session duration is declining (from 23 minutes to 17 minutes over 6 months) - Enterprise churn rate is 8.2% quarterly (industry benchmark: 5%) - NPS score dropped from 42 to 31

Stakes and Constraints

NovaMind must improve completion rates to 50% within 6 months to retain their largest enterprise client (\$4.2M annual contract) and hit their Series C growth targets.

Technical constraints: - Recommendations must be generated in under 200ms - The system must work for new users with no history (cold start) - Content library: 12,000 microlearning modules across 840 topics - User interactions: ~8M per day - Model must be explainable for enterprise compliance

Why Policy Gradient Methods?

This problem is fundamentally a **sequential decision-making** problem under uncertainty. At each step, the system must choose which content to show next based on the learner's current state. The "reward" is delayed -- we only know if the content sequence was effective after the learner completes (or abandons) their session.

Value-based methods (like DQN) struggle here because: - The action space is large (12,000 modules to choose from) - Actions need a probability distribution for exploration, not a single best action - The policy must naturally handle stochastic exploration

Policy gradient methods are the right solution because they directly output a probability distribution over the 12,000 modules, naturally handle the large action space, and allow controlled exploration through the softmax temperature.

Section 2: Technical Problem Formulation

Problem Type: Sequential Recommendation as a Markov Decision Process

We frame the adaptive content recommendation problem as an MDP where: - **States** represent the learner's current knowledge state - **Actions** represent content module recommendations - **Rewards** capture learning effectiveness - **Transitions** model how the learner's state changes after interacting with content

Input/Output Specification

State vector $s_t \in \mathbb{R}^{256}$: - Learner embedding (128-dim): encodes historical performance, content preferences, and learning velocity - Session context (64-dim): current session duration, number of modules completed, time of day, device type - Knowledge state (64-dim): estimated mastery across 64 topic clusters

Action $a_t \in \{1, 2, \ldots, 12000\}$: - Index of the microlearning module to recommend next

Reward r_t : - Module completion: +1.0 if the learner completes the module, 0.0 otherwise - Quiz performance: +0.5 * (quiz score / 100) if the module contains a quiz - Engagement bonus: +0.3 if the learner spends at least 80% of expected time on the module - Negative signal: -0.5 if the learner skips the module within 30 seconds

Mathematical Foundation

Policy parameterization:

We use a neural network policy $\pi_\theta(a|s)$ that outputs a probability distribution over all 12,000 modules. However, directly computing softmax over 12,000 actions is computationally expensive. We use a two-stage approach:

Stage 1 (Candidate generation): A lightweight model retrieves the top-K=100 candidate modules based on topic relevance and difficulty matching.

Stage 2 (Policy scoring): The policy network π_θ scores only the K=100 candidates:

$$\pi_\theta(a_i|s) = \frac{e^{h_\theta(s,a_i)}}{\sum_{j=1}^{K} e^{h_\theta(s,a_j)}}$$

where $h_\theta(s, a_i)$ is the preference score for module a_i given state s .

Let us plug in simple numbers. Suppose K=3 candidate modules with scores $h(a_1) = 3.2$, $h(a_2) = 2.8$, $h(a_3) = 1.5$:

$$e^{3.2} = 24.53, \quad e^{2.8} = 16.44, \quad e^{1.5} = 4.48$$

$\text{sum} = 45.45$

$$\pi(a_1) = 0.54, \quad \pi(a_2) = 0.36, \quad \pi(a_3) = 0.10$$

Module a_1 has the highest probability but a_2 and a_3 still have non-trivial probability, allowing exploration.

Performance measure:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T-1} \gamma^t r_t \right]$$

where T is the session length (variable, up to 20 interactions) and $\gamma = 0.95$.

Policy gradient update with baseline:

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot (G_t - V_\phi(s_t))\right]$$

We use an Actor-Critic architecture where: - **Actor** (π_θ): Recommends content - **Critic** (V_ϕ): Estimates expected session return

Loss Function

The total loss has three terms:

$$\mathcal{L} = \mathcal{L}_{\text{actor}} + c_1 \mathcal{L}_{\text{critic}} + c_2 \mathcal{L}_{\text{entropy}}$$

Term 1 -- Actor loss: Negative of the policy gradient objective

$$\mathcal{L}_{\text{actor}} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T_i-1} \log \pi_\theta(a_t^{(i)}|s_t^{(i)}) \cdot A_t^{(i)}$$

Justification: This directly optimizes the policy to increase the probability of actions that led to higher-than-expected returns and decrease the probability of those that led to lower-than-expected returns.

Term 2 -- Critic loss: Mean squared error between predicted and actual returns

$$\mathcal{L}_{\text{critic}} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T_i-1} (V_\phi(s_t^{(i)}) - G_t^{(i)})^2$$

Justification: Training the critic to accurately predict returns provides a low-variance baseline for the actor, reducing gradient noise.

Term 3 -- Entropy bonus:

$$\mathcal{L}_{\text{entropy}} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T_i-1} H(\pi_\theta(\cdot|s_t^{(i)}))$$

Justification: Prevents the policy from collapsing to always recommending the same few modules. Controlled exploration is critical in an educational setting where diverse content exposure benefits learning.

Coefficients: $c_1 = 0.5$, $c_2 = 0.01$.

Evaluation Metrics

Metric	Definition	Target
Completion Rate	% of recommended modules fully completed	> 50%
Session Return	Average cumulative reward per session	> 8.0

Metric	Definition	Target
Knowledge Gain	Pre-post test score improvement	> 15 pp
Exploration Ratio	% of unique modules recommended in 1000 sessions	> 40%

Baseline Method

Rule-based recommender: The current production system that assigns content based on pre-test scores using a fixed decision tree. This achieves 34% completion rate and 5.2 average session return.

Section 3: Implementation Notebook Structure

3.1 Data Generation and Preprocessing

```
# TODO: Implement the data generator for simulated learner interactions

class LearnerEnvironment:
    """
    Simulated environment for adaptive content recommendation.
    Models learner behavior as an MDP.
    """
    def __init__(self, num_modules=100, state_dim=32, num_topics=10):
        """
        Args:
            num_modules: number of content modules available
            state_dim: dimension of the state vector
            num_topics: number of topic clusters
        """
        # ============ TODO ============
        # Initialize the environment:
        # 1. Create module difficulty levels (uniform 0-1)
        # 2. Create module-topic assignments
        # 3. Initialize learner knowledge state
        # ==============================
        pass

    def reset(self):
        """Reset the environment for a new session."""
        # ============ TODO ============
        # Return initial state vector
        # ==============================
        pass

    def step(self, action):
        """
        Execute one step: learner interacts with recommended module.

        Returns: next_state, reward, done, info
        """
        # ============ TODO ============
        # 1. Check if module difficulty matches learner level
        # 2. Compute completion probability
        # 3. Compute reward based on completion and engagement
        # 4. Update learner knowledge state
        # 5. Return (next_state, reward, done, info)
        # ==============================
        pass
```

3.2 Exploratory Data Analysis

```
# TODO: Analyze the simulated learner data

def analyze_learner_data(env, num_episodes=1000):
    """
    Collect episodes using random policy and analyze:
    - Distribution of episode lengths
    - Reward distribution
    - Completion rates by difficulty
    - Knowledge gain distribution
    """
    # ============ TODO ============
    # 1. Collect episodes with random actions
    # 2. Plot episode length histogram
    # 3. Plot reward distribution
    # 4. Plot completion rate vs module difficulty
    # 5. Compute summary statistics
    # ==============================
    pass
```

3.3 Baseline Model

```
# TODO: Implement the rule-based baseline recommender

class RuleBasedRecommender:
    """
    Baseline recommender that matches modules to learner level.
    Always picks the module closest to the learner's current mastery level.
    """
    def recommend(self, state, candidate_modules):
        """
        Args:
            state: learner state vector
            candidate_modules: list of module indices

        Returns:
            action: index of recommended module
        """
        # ============ TODO ============
        # 1. Extract learner mastery from state
        # 2. Find module with closest difficulty match
        # 3. Return module index (no exploration)
        # ==============================
        pass
```

3.4 Policy Gradient Model Architecture

```
# TODO: Implement the Actor-Critic content recommender

class ContentRecommenderActor(nn.Module):
    """
    Policy network for content recommendation.
    Maps learner state to a distribution over candidate modules.
    """
    def __init__(self, state_dim, hidden_dim=256, num_candidates=100):
        """
        Architecture:
        - State encoder: state_dim -> hidden_dim -> hidden_dim
        - Module scorer: takes encoded state + module embedding, outputs scalar score
        - Softmax over candidate scores
        """
        super().__init__()
        # ============ TODO ============
        # Build the actor network
        # ==============================
        pass

    def forward(self, state, candidate_embeddings):
        """
```

```
        Args:
            state: (batch, state_dim)
            candidate_embeddings: (batch, K, embed_dim)

        Returns:
            action_probs: (batch, K) probability distribution over candidates
        """
        # ============ TODO ============
        pass

class ContentRecommenderCritic(nn.Module):
    """Value network: estimates expected session return from current state."""
    def __init__(self, state_dim, hidden_dim=256):
        super().__init__()
        # ============ TODO ============
        pass

    def forward(self, state):
        # ============ TODO ============
        pass
```

3.5 Training Loop

```
# TODO: Implement the Actor-Critic training loop

def train_content_recommender(env, actor, critic, num_episodes=5000,
                               lr_actor=1e-3, lr_critic=1e-3,
                               gamma=0.95, entropy_coeff=0.01):
    """
    Train the content recommender using Actor-Critic with entropy bonus.

    Returns:
        reward_history, completion_rate_history, exploration_history
    """
    # ============ TODO ============
    # 1. Initialize optimizers
    # 2. For each episode:
    #    a. Reset environment
    #    b. Collect full episode (states, actions, rewards, log_probs)
    #    c. Compute returns G_t
    #    d. Compute advantages A_t = G_t - V(s_t)
    #    e. Compute actor loss, critic loss, entropy bonus
    #    f. Update both networks
    # 3. Track metrics: rewards, completion rates, exploration ratio
    # ==============================
    pass
```

3.6 Evaluation

```
# TODO: Evaluate the trained recommender against baseline

def evaluate_recommender(env, model, num_episodes=500):
    """
    Evaluate without exploration (greedy policy).

    Returns:
        completion_rate, avg_session_return, avg_knowledge_gain, unique_modules_recommended
    """
    # ============ TODO ============
    # 1. Run episodes with greedy action selection
    # 2. Compute all evaluation metrics
    # 3. Return metrics dictionary
    # ==============================
    pass
```

3.7 Error Analysis

```
# TODO: Analyze failure modes
```

```python
def error_analysis(env, model, num_episodes=200):
    """
    Identify when and why the recommender fails.

    Analyze:
    - Which learner profiles get low rewards
    - Which modules are over/under-recommended
    - Session length distribution for completed vs abandoned sessions
    - Knowledge gain vs session return correlation
    """
    # ============ TODO ============
    pass
```

3.8 Deployment Simulation

```python
# TODO: Simulate A/B test deployment

def simulate_ab_test(env, baseline_model, pg_model, num_users=1000, sessions_per_user=5):
    """
    Simulate an A/B test between rule-based and policy gradient recommenders.

    Args:
        env: learner environment
        baseline_model: rule-based recommender
        pg_model: trained policy gradient recommender
        num_users: number of simulated users
        sessions_per_user: sessions per user

    Returns:
        A/B test results with statistical significance
    """
    # ============ TODO ============
    # 1. Split users 50/50
    # 2. Run sessions for each group
    # 3. Compute metrics for each group
    # 4. Run statistical significance test (t-test)
    # 5. Report lift and p-value
    # ==============================
    pass
```

3.9 Ethics and Fairness

```python
# TODO: Analyze fairness across learner demographics

def fairness_analysis(env, model, num_episodes=1000):
    """
    Check if the recommender performs equally well across:
    - Different initial knowledge levels (beginner, intermediate, advanced)
    - Different learning speeds (fast, medium, slow)
    - Different session lengths (short, medium, long)

    Report any disparities.
    """
    # ============ TODO ============
    # 1. Create learner cohorts by knowledge level
    # 2. Run episodes for each cohort
    # 3. Compare completion rates and knowledge gains
    # 4. Flag any cohort with >10% performance gap from the best
    # ==============================
    pass
```

Section 4: Production and System Design Extension

System Architecture

```
                                         ┌──────────────────────┐
                                         │  Content Management  │
                                         │       System         │
                                         └──────────┬───────────┘
                                                    │
┌───────────┐     ┌───────────────┐      ┌──────────▼───────────┐     ┌──────────────┐
│  Learner  │────▶│  API Gateway  │────▶│  Recommendation      │────▶│  Content     │
│  Client   │◀────│  (< 200ms)    │◀────│  Service             │     │  Delivery    │
└───────────┘     └───────────────┘      │                      │     └──────────────┘
                                         │  ┌────────────────┐  │
                                         │  │ Candidate Gen  │  │
                                         │  │ (Stage 1)      │  │
                                         │  └───────┬────────┘  │
                                         │          │           │
                                         │  ┌───────▼────────┐  │
                                         │  │ Policy Network │  │
                                         │  │ (Stage 2)      │  │
                                         │  └────────────────┘  │
                                         └──────────┬───────────┘
                                                    │
                                         ┌──────────▼───────────┐
                                         │  Event Streaming     │
                                         │  (Kafka)             │
                                         └──────────┬───────────┘
                                                    │
                                ┌───────────────────┼───────────────────┐
                                │                   │                   │
                    ┌───────────▼───────┐  ┌────────▼─────────┐ ┌───────▼───────┐
                    │  Feature Store    │  │ Training Pipeline│ │  Monitoring   │
                    │  (Redis)          │  │ (Batch, nightly) │ │  (Grafana)    │
                    └───────────────────┘  └──────────────────┘ └───────────────┘
```

API Design

```
POST /api/v1/recommend
Request:
{
  "user_id": "u_12345",
  "session_id": "s_67890",
  "context": {
    "device": "mobile",
    "time_of_day": "evening",
    "session_duration_so_far_seconds": 420
  }
}

Response (< 200ms):
{
  "module_id": "m_789",
  "module_title": "Introduction to Linear Regression",
  "confidence": 0.72,
  "alternatives": [
    {"module_id": "m_456", "confidence": 0.18},
    {"module_id": "m_123", "confidence": 0.10}
  ],
  "explanation": "Selected based on your recent progress in statistics fundamentals."
}
```

Model Serving

- **Inference framework:** ONNX Runtime for model inference (sub-10ms)
- **Candidate generation:** Pre-computed nearest-neighbor index (FAISS) for top-100 retrieval (sub-5ms)

- **Feature computation:** Redis feature store with pre-computed learner embeddings (sub-2ms)
- **Total latency budget:** < 50ms (well within the 200ms API requirement)
- **Throughput:** 10,000 recommendations/second on a single GPU instance

Monitoring and Drift Detection

Real-time metrics (Grafana dashboards): - Recommendation latency (P50, P95, P99) - Session completion rate (rolling 1-hour window) - Average session return (rolling 1-hour window) - Policy entropy (should not collapse below threshold)

Drift detection: - Feature drift: Monitor distribution of learner state vectors (KL divergence) - Reward drift: Detect shifts in completion rate or engagement signals - Action drift: Monitor whether the policy concentrates on too few modules

Alert thresholds: - Completion rate drops below 40% for > 1 hour - Policy entropy drops below 1.0 (policy collapse) - P99 latency exceeds 150ms - Any single module recommended > 5% of the time

A/B Testing Framework

- **Traffic split:** 90% current model / 10% new model initially
- **Ramp-up schedule:** 10% -> 25% -> 50% -> 100% over 4 weeks
- **Primary metric:** 7-day learner completion rate
- **Guardrail metrics:** Session duration, skip rate, NPS score
- **Statistical framework:** Sequential testing with alpha-spending function to enable early stopping

CI/CD Pipeline

1. **Data validation:** Check for feature drift in training data
2. **Model training:** Nightly retraining on the last 7 days of interaction data
3. **Offline evaluation:** Compare new model against current production model on held-out sessions
4. **Shadow deployment:** Run new model in shadow mode, compare recommendations without serving them
5. **Canary release:** Serve to 1% of traffic, monitor all metrics
6. **Full rollout:** Gradual traffic ramp-up with automatic rollback on metric degradation

Cost Estimation

Component	Monthly Cost
GPU inference (2x A10G)	\$2,400
Redis feature store	\$800

Component	Monthly Cost
Kafka event streaming	\$600
Training compute (nightly)	\$1,200
Monitoring infrastructure	\$400
Total	**\$5,400/month**

At 2.3M users and \$18/user/month, infrastructure cost is 0.013% of revenue — negligible.

References

1. Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). "Policy Gradient Methods for Reinforcement Learning with Function Approximation." NeurIPS.
2. Williams, R. J. (1992). "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." Machine Learning.
3. Ie, E., et al. (2019). "SlateQ: A Tractable Decomposition for Reinforcement Learning with Recommendation Sets." AAAI.
4. Chen, M., et al. (2019). "Top-K Off-Policy Correction for a REINFORCE Recommender System." WSDM.