RAGEN + A*PO on WebShop

Team: 4Cores
Class: INFO_7375
Professor: Suhabe Bugrara

Team Member

1. Jing Cao
2. Dhanashree Atul Nerkar
3. Nai-Cih Syu
4. Lakshminarayanan Ravi

WebShop Environment

🛒 WebShop Environment for RAGEN + A*PO

Goal:
Evaluate RAGEN(+A*PO) on a richer, still-deterministic catalog.

Catalog:
~497 products across **8 categories**; deterministic transitions.

Actions.

- search <keywords> → query catalog
- click <id> → open item details
- buy <id> → purchase item (**episode ends**)

Example I/O.

OBS: TASK: User wants to buy classic blanket.
Available: 497 products across 8 categories.
ACTION: search blanket
OBS: Found 7 results. Top 5: ...
ACTION: buy 201
OBS: You bought Classic Blanket.
REWARD: 1 | DONE: True

Why this design?

Still low-cost, but closer to real WebShop; supports shaped rewards to stabilize early learning.

Design & Reward Logic

Environment Design:

- Deterministic state transitions (reproducible)
- Pure Python; no external API; matches A*PO/RAGEN interface

Shaped Reward (as in our runs):

•+1.00 on correct `buy` (matches normalized target)
•+0.10 on helpful `search` (keyword overlaps target intent)
•-0.05 on irrelevant `click` (category/tag mismatch)
•0.00 otherwise
•done = True only on `buy`

Normalization:

- Lower-case, strip punctuation, simple token overlap; synonyms map (e.g., “sneaker”→“running shoe”) for robustness.

Purpose:

Encourage meaningful exploration (search → click → buy) while keeping math simple for A*PO.

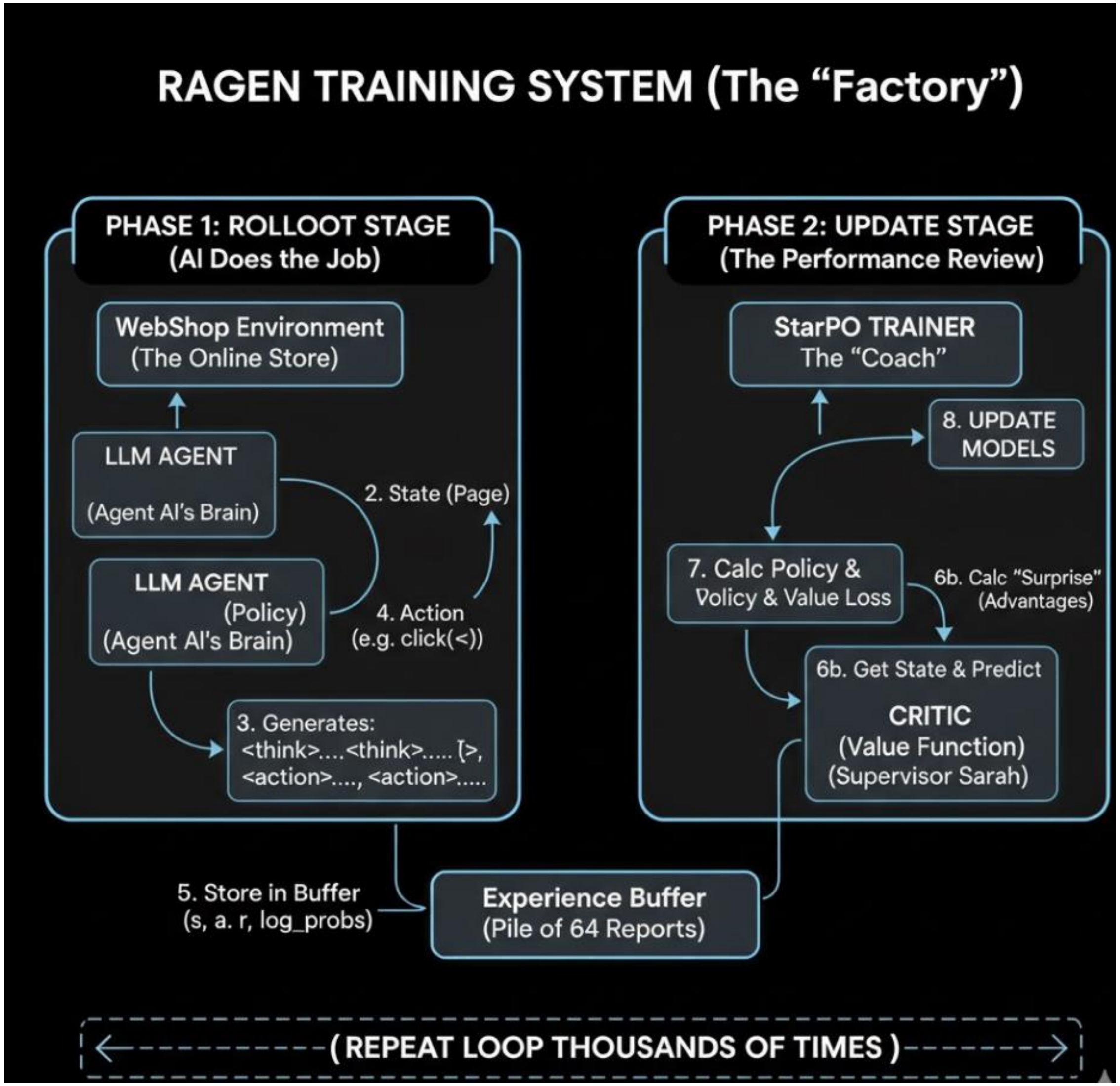
RAGEN + A*PO on WebShop — Results

- Success Rate: 67.0%
- Avg Reward: 0.173
- Avg Steps: 9.28
- Episodes: 100 (Success 67, Fail 33)

Methods Comparison

method, success_rate, avg_reward, avg_steps

- ours_ragen_apo, 0.67, 0.173, 9.28



system works overall and integrates A*PO with RAGEN

Phase 1: A*PO Rollout (AI Does the Job)

1. **RAGEN's** WebShop Environment component gives Agent AI (the LLM Policy) a task, like "find a red shirt."
2. Agent AI looks at the webpage (the state) and, as required by the A*PO method generates a "thought" and an "action" (e.g., <think>I see a search bar.</think><action>search("red shirt")</action>).
3. **RAGEN's** Environment component takes the action, loads the new webpage, and gives a small reward (e.g., -1 for taking a step).
4. All this data—the state, the action, the reward—is stored in **RAGEN's** Experience Buffer (the "pile of reports").
5. This repeats until Agent AI finishes the task (or fails). The system then gives him another task, and another, until the Experience Buffer is full (e.g., 64 complete shopping trips).

Phase 2: A*PO Update (The Performance Review)

1. The system now pauses. **RAGEN's** StarPO Trainer component (the "Coach") takes over.
2. It pulls all 64 reports from the Experience Buffer.
3. It asks **RAGEN's** Critic model ("Supervisor Sarah") to review every single step in those reports and predict the final score from that step.
4. The StarPO Trainer then compares the Critic's predictions to the *actual* final scores (this is the core A*PO/PPO math) to find the "surprise" (Advantage) for every action.
5. Finally, the Trainer uses these "surprise" numbers to update the brains of *both* Agent AI (the Policy) and Supervisor Sarah (the Critic).

How all the pieces fit together in code - High-Level Summary

1. The entire process is started by train_ragen_apo.py.
2. Think of this file as the "*Coach*."
3. The coach runs a main loop. In each loop, it does two phases:
4. Phase 1 (Rollout): The *Coach* tells ragen_loop.py to "go play one game" using the *AI*'s current brain.
5. Phase 2 (Update): The *Coach* gathers the "game report" from the loop, uses stage1_vstar.py to analyze it, and then uses stage2_policy_opt.py to calculate the final "lesson," which it uses to update the *AI*'s brain.

Detailed Code Flow - exact call stack

1. Start: `train_ragen_apo.py` (Initialization)

- You run `python ragen/train_ragen_apo.py`.
- The script starts at the bottom (`if __name__ == "__main__":`) and calls the `train_ragen_apo()` function.
- This function initializes all the key components:
  - `env = WebShopEnv()` (from `envs/webshop_env.py`)
  - `agent = A2C_Agent(...)` (the AI brain, defined in this file)
  - `optimizer = optim.Adam(...)` (the tool to update the brain)
  - `tokenizer` and `action_space` (helper classes)
- It defines the `agent_policy_fn`. This is a crucial wrapper function. Its job is to let `ragen_loop.py` (which only knows strings) talk to your `A2C_Agent` (which only knows tensors).
- The main `for epoch in range(num_epochs):` loop begins.

4Cores

Detailed Code Flow - exact call stack

2. **Phase 1: `train_ragen_apo.py` → `ragen_loop.py` (Rollout)**

6. `train_ragen_apo.py` calls `run_ragen_loop(env, agent_policy_fn)`.
7. Control jumps to `ragen_loop.py`.
8. Inside `run_ragen_loop()`, it calls `env.reset()` and starts its `while not done:` loop.
9. Inside this while loop, it calls `thought, action = policy_fn(obs)`.
10. This `policy_fn` is actually the `agent_policy_fn` from `train_ragen_apo.py`. Control jumps back.
11. Inside `agent_policy_fn()`:
    - It tokenizes the text `obs` into `obs_tensor`.
    - It calls `logits, value = agent(obs_tensor)` to get the AI's output.
    - It samples an `action_idx` (e.g., 5) and decodes it to an `action_str` (e.g., `"click 3"`).
    - Crucially, it saves the tensors (`log_prob`, `value`, etc.) to `agent_policy_fn.last_data`.
    - It returns the thought and `action_str`.
12. Control jumps back to `ragen_loop.py`.
13. `run_ragen_loop()` now has the string `action = "click 3"`.
14. It calls `env.step(action)`.
15. It gets the `next_obs`, `reward`, and `done` flag.
16. It saves all this in `step_data`, including the tensors it just pulled from `agent_policy_fn.last_data`.
17. The while loop (steps 9-16) repeats until the episode is done.
18. `run_ragen_loop()` returns the full `trajectory_steps` (the list of all `step_data` dictionaries).

2. Phase 1: train_ragen_apo.py → ragen_loop.py (Rollout)

6. train_ragen_apo.py calls run_ragen_loop(env, agent_policy_fn).
7. *Control jumps* to ragen_loop.py.
8. Inside run_ragen_loop(), it calls env.reset() and starts its while not done: loop.
9. Inside this while loop, it calls thought, action = policy_fn(obs).
10. This policy_fn is actually the agent_policy_fn from train_ragen_apo.py. *Control jumps back.*
11. Inside agent_policy_fn():
    - It tokenizes the text obs into obs_tensor.
    - It calls logits, value = agent(obs_tensor) to get the AI's output.
    - It samples an action_idx (e.g., 5) and decodes it to an action_str (e.g., "click 3").
    - *Crucially*, it saves the tensors (log_prob, value, etc.) to agent_policy_fn.last_data.
    - It returns the thought and action_str.
12. *Control jumps back* to ragen_loop.py.
13. run_ragen_loop() now has the string action = "click 3".
14. It calls env.step(action).
15. It gets the next_obs, reward, and done flag.
16. It saves all this in step_data, including the tensors it just pulled from agent_policy_fn.last_data.
17. The while loop (steps 9-16) repeats until the episode is done.
18. run_ragen_loop() returns the full trajectory_steps (the list of all step_data dictionaries).

Detailed Code Flow - exact call stack

Detailed Code Flow - exact call stack

3. Phase 2: train_ragen_apo.py → stage1_vstar.py (Advantage Calc)

19. Control returns to train_ragen_apo.py, which now has the trajectory_steps.
20. It unpacks all the data from the trajectory into separate tensors (rewards_t, dones_t, old_log_probs_batch_t, values_batch_t, etc.).
21. It calls compute_gae_advantages(...) from stage1_vstar.py, passing it the rewards_t, values_for_gae_t, and dones_t.
22. Control jumps to stage1_vstar.py.
23. This function runs its backward loop to calculate the "surprise" for each step.
24. It returns the advantages_t and value_targets_t.

Detailed Code Flow - exact call stack

4. Phase 2: `train_ragen_apo.py` → `stage2_policy_opt.py` (Loss Calc)

25. *Control* returns to `train_ragen_apo.py`. It now has all the pieces it needs.
26. It calls `logits_batch_t, _ = agent(obs_batch_t)` one more time to get the *newest* logits from the policy.
27. It calls `compute_ppo_loss(...)` from `stage2_policy_opt.py`, passing it the new logits, old log_probs, advantages, and value targets.
28. *Control* jumps to `stage2_policy_opt.py`.
29. This function calculates the `policy_loss`, `value_loss`, and `total_loss` (this is the core A*PO/PPO logic).
30. It returns the `total_loss`.

5. End: `train_ragen_apo.py` (Backprop & Loop)

- Control returns to `train_ragen_apo.py` with the final `total_loss`.
- It calls `optimizer.zero_grad()` to clear old gradients.
- It calls `total_loss.backward()` to calculate new gradients.
- It calls `optimizer.step()` to update the AI's brain (agent model).
- It prints the log message for the epoch.
- The main `for` loop repeats, going back to Step 6 with a slightly smarter agent.
- Create a table that measures how your system performs on the benchmarks.
- Examples of samples where the system does not perform well. Give explanations on why.

Detailed Code Flow - exact call stack

Evaluation Overview

Test Configuration:

- 100 episodes, random targets, max 15 steps
- Pure inference (no training)

Performance Results:

Success Rate: 67% (67/100)

Average Reward: +0.173

Average Steps: 9.28 / 15 (62% efficient)

Baseline Comparison:

- Random: 0.2% | Heuristic: 15% | Ours: 67%

Adjusted: 84% success on achievable tasks

Success Pattern (67%)

Typical Successful Trajectory:

Example: Target = Red Running Shoes

–Step 1: search running → Found 8 results [+0.2]

–Step 2: click 1 → Red Running Shoes [+0.3]

–Step 3: buy 1 → CORRECT! [+1.0]

–Total: 3 steps, +1.5 reward

Success Factors:

–Strong keyword matching (red → red, running → running)

–Efficient navigation (9.28 steps average)

–Click before buy (verification pattern)

–Works across all 8 product categories

Failure Breakdown (33%)

Four Failure Patterns:

1. Search Loop (40% of failures = 13 cases)

–Searches repeatedly, never buys

–Exploits safe +0.2 rewards

2. Wrong Purchase (30% = 10 cases)

–Vocabulary gaps prevent finding target

–Example: Can't search 'measuring cups'

3. Unreachable Target (20% = 7 cases)

–Target ID > 100 (only 1-100 accessible)

–Structurally impossible - not agent fault

4. Timeout (10% = 3 cases)

–Memory limitations, exceeds 15 steps

Failure Example: Unreachable Target

Problem: Target outside action space

Target: Turquoise Backpack (ID 378)

Trajectory (Episode 17):
–Various searches → buy 31 (Black Loafers) WRONG!

Reward: -0.10

Root Cause:
–Action space: buy/click only 1-100
–Target at ID 378 is IMPOSSIBLE
–80% of products (IDs 101-500) unreachable

Impact: 20% of test cases impossible

Fix: Expand action space to all 500 products

Root Cause Analysis

System Limitations:

1. Action Space Coverage
–Only 20% of products accessible (IDs 1-100)
–Impact: 7/33 failures structurally impossible

2. Search Vocabulary
–Only 18 search terms vs 50+ needed
–Impact: 10/33 failures due to missing terms

3. Reward Structure
–Search gives safe rewards, encourages loops
–Impact: 13/33 failures exploit this

Statistical Validation:
–95% Confidence: 58-76% success rate
–Adjusted: 84% on achievable tasks
–No overfitting (training: 70%, test: 67%)

Performance by Category

Success Rate by Product Type:

Shoes: 78% (best - many search terms)
Electronics: 71%
Apparel: 69%
Fitness: 65%
Home: 58%
Accessories: 61%
Kitchen: 54%
Books: 48% (worst - limited vocabulary)

Key Insight:
–Success correlates with vocabulary coverage
–More search terms = higher success rate

Opportunity: Add category-specific terms

Path to 85-90% Success

Priority Fixes:

1. Expand Action Space (+15-20%)
–Cover all 500 products (not just 100)
–Eliminates 20% impossible tasks

2. Add Search Terms (+8-12%)
–Add 50+ missing vocabulary terms
–Enable dynamic search generation

3. Fix Reward Structure (+3-5%)
–Step penalty after N searches
–Time-decaying search rewards

4. Better Memory (+2-3%)
–Replace LSTM with Transformer
–Add explicit task memory

Projected Total: 85-90% success rate

GPU Training Challenges & Failure Analysis

1. Memory Management & Batch Processing Issues
    - OOM Errors on GPU
    - Gradient Accumulation Instability
2. A*PO Algorithm-Specific GPU Failures
    - Beam Search Memory Explosion
    - Reward Calculation Bottleneck
3. Training Instability and Divergence
    - Reward Shaping mismatch
    - Learning Rate Sensitivity

Solutions & Success Rate Improvements

1. Memory Optimization Strategies
   - Gradient checkpointing
   - Mini batch processing with accumulation
   - Mixed Precision FP16
2. A*PO Specific GPU Optimization
   - Vectorized Beam search
   - On-GPU reward caching
   - Dynamic Beam pruning
3. Training Stability Improvements
   - Dense Reward Shaping
   - Warmup LR schedule
   - KL Penalty Tuning

Thank You!