Learning the Universal Matrix: A Rule-Guided Reinforcement Learner

Monada Dominion Research

October 2025

Abstract

We present a reinforcement-learning (RL) control stack that reproduces the twelve-step Universal Matrix construction introduced in *Universal Matrix: A Least-Action Cartesian Construction Protocol Embodying Occam's Razor and the Principle of Least Action in Geometric Construction* [1]. The agent follows an explicitly validated action grammar, integrates deterministic geometric rules into its transition dynamics, and learns to synthesize blueprint-quality renderings without exposing implementation details. We summarize the design motivation, environment interface, curriculum schedule, reward shaping, and empirical behavior, emphasizing how each component encodes the Occam-style minimality principles underlying the original construction.

1 Introduction

The Universal Matrix protocol described in *Universal Matrix: A Least-Action Cartesian Construction Protocol Embodying Occam's Razor and the Principle of Least Action in Geometric Construction* [1] specifies a twelve-step, least-action blueprint for arriving at a self-reflective hexagonal pattern. Our reinforcement-learning (RL) agent replays that deterministic script inside a simulator whose transition rules, validators, and rewards embody the same minimality constraints. The goal is not to publish executable code, but to document the logic that makes the policy successful so that the method can be reviewed, reproduced, or re-implemented without exposing proprietary assets.

The agent operates in a symbolic geometry sandbox where every admissible action corresponds to a primitive from the Occam protocol. Rather than discovering geometry from scratch, learning focuses on sequencing: the policy must select the right primitive, supply consistent arguments, and respect gating rules that disallow shortcuts. When those conditions hold, blueprint-quality renderings emerge from the same chain that a human geometer would follow.

This paper captures the system architecture, the step-wise geometric executors, validator stacks, reward model, curriculum, and monitoring pipeline. Each section connects software components back to the Occam principles of least action, determinism, inheritance, and emergent self-reflection. The resulting description is self-contained and sufficiently detailed for Overleaf compilation, yet abstracted away from any release of source code.

2 System Architecture

2.1 Component Overview

The agent stack is organized as a small collection of cooperating modules. Table 1 summarizes the responsibilities and the files where each behaviour is encoded. File paths are given relative to the private repository and can be inspected without disclosing implementation internals.

oprule Component	Responsibility
extttum_env.UMEnv	Gymnasium-compatible environment orchestrating state, rewards, and rendering
extttenv/steps/*.py	Deterministic executors for Steps 1–12 of the Occam protocol
extttenv/mixins/validation.py	Geometric validators for convexity, intersections, and polygon cycles
extttenv/mixins/cost.py	Least-action cost accounting and milestone targets
exttttoolkit/primitives.py	Geometry helpers (points, line intersections, plotting utilities)
extttttrain.py	PPO training harness using `stable_baselines3` with checkpoints
extttcurriculum_trainer.py	Stage-by-stage trainer passing intermediate states forward
extttmemory.py	Persistent database of failed state–action pairs
exttttest_runner.py	Deterministic smoke tests with interactive visualization

Table 1: Core modules governing the Universal Matrix reinforcement learner.

2.2 Execution Flow

Episodes begin with `UMEnv.reset()`, which optionally randomizes the initial anchor segment while seeding reproducible runs. Each time step the policy emits a five-dimensional action vector; the environment coerces or rejects invalid primitives before dispatching to the corresponding step executor. Validators inspect structural and geometric invariants after every mutation, rewards and penalties are computed, and the observation vector is regenerated. When the episode terminates the environment caches renderings and logs diagnostics to `episode_logs.txt` for later review.

3 Environmental Abstractions

3.1 Action Alphabet

The environment exposes a `MultiDiscrete([13, 30, 30, 30, 30])` action space. The first index chooses among thirteen primitive handlers aligned with the twelve Occam steps plus a dedicated center-marking primitive; the four auxiliary indices identify point labels when the primitive requires operands. Each primitive mirrors a physical operation from the research paper, and any macro expands into deterministic primitive sequences before the agent observes their effects. The available primitives can be grouped as follows:

- Point placement via directed length transfers with lexical tie-breaking.
- Segment drawing between previously instantiated points.
- Intersection queries that materialize concurrency witnesses.
- Removal operations that prune auxiliary scaffolding under conservation rules.
- Macro wrappers (e.g., rope locus traversal) implemented as deterministic action scripts.

Each primitive is wrapped in a class inside `env/steps`. Step handlers expose a `handle` method that verifies preconditions, mutates the symbolic state, and returns shaped rewards together with step-specific diagnostic data. Invalid choices never advance the simulator: instead, `UMEnv` coerces the primitive index toward the nearest valid option when strict gating is enabled, applying a small penalty so that policies learn to respect the grammar autonomously.

3.2 Observation Encoding

A symbolic workspace tracks points, segments, and derived relations. The observation returned to the agent concatenates coordinate slots and global features:

- **Point buffer**: a fixed matrix of up to 30 point coordinates ordered lexicographically by label.
- **Structural metrics**: counts of points and edges, a convex-quadrilateral indicator, and four local angles useful during early steps.

- **Noise injection**: optional Gaussian perturbations controlled by the `UM_OBS_NOISE` environment variable for robustness experiments.

The agent therefore receives continuous feedback about progress toward each milestone without ever seeing raw mesh objects. Lexicographic ordering ensures consistency between observations and the deterministic tie-breaking rules from the Occam paper, enabling direct comparisons between manual and learned traces.

3.3 Primitive Gating and Assistance

Allowed primitives depend on curriculum stage and milestone set. When `UM_STRICT_GATING=1` (default), the environment restricts choices to the minimal admissible subset for the current stage—for example, Step 8 actions become available only after the hexagon and its center are validated. Assistance toggles, exposed via environment variables, modulate how much autonomy is given to the learner:

- `UM_AGENT_CHOICES`: lets the policy choose operand indices; disabling it replays canonical operands for curriculum bootstrapping.
- `UM_ASSISTED_STEPS`: keeps macro steps deterministic for smoke testing while leaving primitive selection to the agent.
- `UM_RANDOM_RESET`: samples random similarity transforms for the initial configuration, building invariance to translation, rotation, and scale.

All toggles default to the exploratory configuration used during training and can be frozen for reproducible evaluations.

4 Geometric Step Executors

Each Occam step is implemented by a dedicated handler inside `env/steps`. The handlers share a base class that injects the environment reference and exposes a `handle` function returning $(reward, cost, terminated, info)$. The mapping below highlights the most salient behaviours:

1. **Step 1–2** (`PlacePointStep`, `DrawSegmentStep`): create the base segment using canonical labels and persist it in `networkx` structures.
2. **Step 3** (`ConstructTriangleStep`): enforces non-collinearity via convexity checks before admitting the third point.
3. **Step 4** (`ConstructRhombusStep`): generates the exterior fourth vertex while maintaining convexity, preparing diagonals for the next stage.
4. **Step 5** (`FindIntersectionStep`): records the diagonal intersection E and stores it in `pending_intersection` for later reuse.
5. **Step 6** (`ConstructHexagonStep`): recreates the hexagon by sweeping the conserved length $\lambda = |AE|$, prunes scaffolding through `cleanup_points_and_lines`, and cache the radius token.
6. **Step 7** (`MarkHexagonCenterStep`): redraws all three long diagonals, computes their intersection, and labels the center while leaving perimeter edges untouched.
7. **Steps 8–12** (`Step8SymmetryChords`–`Step12StarOfDavid`): replicate the mirror sequence described in the Occam paper—retaining a canonical diagonal, deriving I_1/I_2 symmetry witnesses, restoring diagonals, building connector intersections V_1/V_2, confirming E by intersecting V lines, and finally drawing the Star of David whose intersections form $R_1, \ldots, R_6$.

Handlers share utilities for edge management and label allocation, guaranteeing that repeated executions remain deterministic. Fallback logic addresses numerical edge cases (for example, alternate connector pairings when computing V points) while still enforcing Occam tie-breakers through lexicographic comparisons.

5 Validation and Rule Enforcement

5.1 Synchronous Validators

Validation logic is mixed into `UMEnv` via `ValidationMixin`. Every action triggers three layers of checks:

1. **Structural validation** confirms operands exist, belong to the current `networkx` graph, and do not exceed configured cardinalities.
2. **Geometric validation** recomputes predicates such as convexity, concurrency, and on-segment membership using analytic geometry helpers (e.g., `segment_intersection_parametric`).
3. **Curriculum validation** verifies that the resulting milestone is legal for the current stage; failing this yields an immediate penalty and termination.

Failures are logged to `episode_logs.txt` with human-readable diagnostics, enabling post-hoc review of learner mistakes.

5.2 Graph-Based Shape Checks

For Step 3 onward, validators search the environment's graph for the minimal cycle compatible with the required polygon. Triangles are detected as cliques; quadrilaterals rely on permutations filtered by vertex degree to confirm convex four-cycles; hexagons are accepted when any six-node cycle appears in the `networkx.cycle_basis`. Intersection validation employs tolerant comparisons to avoid floating-point drift, ensuring that the observed concurrency at E is "seen" rather than constructed.

5.3 Least-Action Cost Accounting

The `CostMixin` computes a dot-and-line based cost metric and compares it to milestone-specific minima. Meeting the minimal cost yields a +15 bonus (applied during key milestones); exceeding it incurs a −10 penalty. Removal primitives are free when they reduce complexity to the minimum required for the next step, encoding the Occam requirement that auxiliary scaffolding be discarded promptly.

5.4 Rule-Based Transition Guards

Step executors include local guards that encode textual rules from the Occam paper. Step 8 retains the lexicographically smallest diagonal, Step 10 derives farthest vertices by computing graph distances of three along the cycle, and Step 11 verifies that `line`(V_1,V_2) re-intersects `line`(I_1,I_2) at E. These guards produce binary outcomes—success advances the milestone, failure leads to a shaped penalty and early termination.

6 Reward Shaping

Rewards and penalties mirror the Occam objectives and are returned directly by step handlers. Typical values include:

- **Step rewards**: foundational steps return 30–100 points when milestones are achieved (e.g., Step 6 grants +100 for the hexagon, Step 12 grants +40 for a valid Star of David).
- **Milestone bonuses**: passing validation with minimal cost yields an additional +15, while exceeding the budget incurs −10.
- **Structural nudges**: earning a convex quadrilateral during Step 4 adds +10; failing convexity or violating gating subtracts −10 and terminates the episode.
- **Invalid action penalties**: coercing an illegal primitive costs −1, invalid milestones result in −25, and unrecoverable geometry (e.g., missing intersections) penalizes −20 to −40 before terminating.

Rewards gradually become sparser as curricula progress. Early stages provide dense shaping to encourage rule compliance; later stages rely on milestone bonuses so the policy learns to plan several primitives ahead without immediate feedback.

7 Training Loop

7.1 Curriculum Strategy

Training progresses across three curricula that mirror the structure of [1]:

1. **Foundation Curriculum**: restricts the action set to Steps 1–6, emphasizing rope-circle macros and length conservation.
2. **Mirror Curriculum**: unlocks Steps 7–12 with mirrored validations and geometric symmetry checks.
3. **Unified Curriculum**: randomizes checkpoints drawn from the entire protocol so the policy learns to recover from partially completed states.

The helper script `curriculum_trainer.py` trains each stage for 50k PPO steps, serializes the resulting environment state (points, graph, milestones), and uses it as the reset template for the next stage. Advancement criteria combine rolling success rates with violation counts, ensuring that the agent does not progress until it consistently satisfies validator expectations.

7.2 Persistent Memory

`memory.py` stores JSON entries keyed by serialized observations and lists of failed actions. During training and smoke testing this memory is queried before executing an action; known failure pairs are skipped to avoid wasting trajectories on repeated mistakes. The log is append-only, deduplicated per state key, and saved after every episode, aligning with the Occam mandate to retain only information that prevents future violations.

7.3 PPO Configuration

The training harness in `train.py` uses the PPO implementation from `stable_baselines3` [2]. The script validates the environment with `check_env`, configures device selection (CPU by default on Apple Silicon), and schedules checkpoint saves every 5,000 steps. Two consecutive runs of 100k timesteps refine the policy within a single curriculum stage before promotion. Hyperparameters remain close to library defaults; the primary inductive bias arises from the environment's deterministic gating rather than algorithmic tuning.

8 Visualization Pipeline

Upon successful completion, the agent emits a declarative geometry log. A rendering toolchain converts the log into vector images that highlight:

- The outer hexagon and inner rotated hexagon.
- Auxiliary scaffolding drawn during intermediate steps.
- Validation checkpoints confirming concurrency, symmetry chords, and skip-one connectors.

Rendering relies on the utilities in `toolkit/primitives.py`. Functions such as `setup_plot` and `plot_labeled_points` convert symbolic states into Matplotlib figures, stamped with UUIDs for traceability. Smoke tests trigger `render()` after every action so that failed episodes can be inspected frame by frame, while batch training stores only milestone snapshots to disk. Because visualization consumes only symbolic data, it reveals the geometry without divulging implementation details.

9 Relation to Occam Principles

The RL system instantiates the four global optimization principles from [1]:

- **Least Action**: validators and penalties discourage superfluous primitives, biasing the agent toward minimal-length trajectories.
- **No Ambiguity**: deterministic tie-breaking and canonical element selection map every state to a unique admissible successor.
- **Inheritance**: conserved length tokens and concurrency witnesses carry forward the essential measurements.
- **Self-Reflection**: the mirror curriculum proves the agent internalizes Steps 7–12 as a scaled replica of Steps 1–6.

10 Empirical Behavior

Training runs converge within a modest number of episodes once the agent enters the unified curriculum. Key observations include:

- Average episode length collapses toward the theoretical minimum as least-action penalties accumulate.
- Violation counts drop sharply after the mirror curriculum unlocks, indicating robust generalization across symmetric step pairs.
- Generated renderings retain the intended auxiliary lines only when diagnostically useful, matching the conservation principle.

Without revealing implementation internals, these metrics demonstrate that the agent faithfully realizes the Occam protocol.

11 Future Directions

Opportunities for extension include curriculum randomization over perturbed initial geometries, probabilistic validators to explore stochastic variants, and meta-learning that induces the validation rules themselves. Each direction maintains the central thesis: minimal, rule-consistent action policies can reproduce intricate constructions when guided by principled rewards.

References

[1] A. Kraskov, "Universal Matrix: A Least-Action Cartesian Construction Protocol Embodying Occam's Razor and the Principle of Least Action in Geometric Construction," ResearchGate, 2025. doi:10.13140/RG.2.2.31033.68967. Available at https://www.researchgate.net/publication/394401785_Universal_Matrix_A_Least-Action_Cartesian_Construction_Protocol_Embodying_Occam%27s_Razor_and_the_Principle_of_Least_Action_in_Geometric_Construction.

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," arXiv preprint arXiv:1707.06347, 2017.