OLA Continuous Learning Development: A Complete Journey

From Catastrophic Forgetting to Breakthrough Discovery

Date: January 15, 2025
Project: Organic Learning Architecture (OLA) - Snake Game Continuous Learning
Duration: Extended research session spanning multiple training runs and 50,000+ episodes

---

Executive Summary

This document chronicles the complete development journey of implementing stable continuous learning in the Organic Learning Architecture (OLA) applied to the Snake game. Over the course of intensive experimentation, we progressed from a system that could learn but not retain (catastrophic forgetting) to achieving a **45% success rate peak with 10-food episodes** - proving that trust-based evolutionary learning can discover and maintain complex game-playing strategies.

The journey involved systematic debugging of trust dynamics, decay mechanisms, elite preservation systems, and ultimately led to the discovery that **relative, context-aware trust protection** is essential for continuous learning. We conclude with a systematic parameter sweep methodology to identify optimal configurations.

Key Achievement: Demonstrated that OLA can learn full Snake gameplay (with self-collision) from scratch in under 5,000 episodes and maintain performance for thousands of episodes - all running at **170-270 episodes per second on CPU**.

Table of Contents

Initial Problem Statement

The Challenge

Implement continuous learning in OLA where the system can:

1. Learn Snake gameplay from random initialization
2. Improve performance over time

3. **Retain learned strategies** without catastrophic forgetting
4. Continue adapting without upper performance bounds

Starting Configuration

- **Population:** 24 genomes (later reduced to 16 for faster iteration)
- **Architecture:** 2-layer MLP (state → 32 hidden → 4 actions)
- **Trust mechanism:** Simple accumulation with fixed decay
- **Learning rate:** 0.05
- **Initial decay:** 0.007 per episode

Initial Observations

Early runs showed promising learning:

- 0% → 40% success in 3,000-5,000 episodes
- Peak performance reaching 6-9 food eaten
- Trust values climbing to 1.99-2.0
- **But then catastrophic collapse back to <10% success**

The fundamental question: **Why can't the system hold what it learns?**

Phase 1: The Decay Crisis

Problem Discovery

Our first major crisis emerged when examining training logs across multiple 5k-episode runs:

Run 1 (Early - Episode 0-5k):

- Best performance: 10 food eaten
- Common: 9 food episodes
- Trust: Genomes peaking at 1.99

Run 2 (Episode 5k-10k):

- Best performance: 8 food (regression)
- Trust: Still 1.99

Run 3 (Episode 10k-15k):

- Best performance: 3 food (catastrophic collapse)
- Trust: Still 1.99

Run 4 (Episode 15k-20k):

- Best performance: 4 food (minimal recovery)

Root Cause Analysis

The issue was **unconditional decay applied to all genomes every episode:**

python

```python
# BROKEN: Original decay implementation
for g in self.genomes:
    g.trust -= 0.007  # Decays EVERY genome EVERY episode
```

Why this failed:

1. Elite genomes discovered good strategies (10 food capability)
2. These elites weren't selected every episode (only ~5-10% of the time)
3. They decayed 0.007 × 100 episodes = 0.7 trust while idle
4. Meanwhile, they only gained trust when selected (rare)
5. **Elite strategies decayed faster than they could be reinforced**
6. Population lost best strategies → performance collapsed

Trust at 1.99 was a red flag - it meant genomes were constantly fighting decay, never stabilizing. They were like climbers on a treadmill that moved faster than they could ascend.

First Fix Attempt: Conditional Decay

python

```python
# Attempt 1: Only decay on failure
if reward < 0:  # Only negative rewards
    g.trust -= 0.007
```

Result: Worse performance

- **Problem:** Most deaths gave small positive rewards from distance shaping
- Wall death = -5.0 reward, but moving toward food then dying = +2.0 reward
- Genomes that consistently failed but "tried" never decayed
- Bad strategies locked in because they had positive rewards

Second Fix Attempt: Threshold-Based Decay

python

```python
# Attempt 2: Decay below performance threshold
if reward < 20:  # Fixed threshold
    g.trust -= 0.007
```

Result: System oscillated, couldn't stabilize

- Threshold of 20 was arbitrary
- Early in training, most episodes gave <20 reward → everyone decayed
- Later in training, threshold became obsolete as population improved
- **Fixed thresholds don't work in continuously evolving systems**

Key Insight #1

> **Continuous learning systems require relative, not absolute, performance measures. Any fixed threshold becomes either too easy or too hard as the population evolves.**

Phase 2: Elite System Experiments

Motivation

After decay crisis, we hypothesized that **explicit elite preservation** was needed. The idea: protect proven strategies from decay entirely.

Elite System v1: Immortal Elites

Design:

python

```
ELITE_REWARD_THRESHOLD = 100  # Fixed threshold to become elite
ELITE_TRUST_THRESHOLD = 1.8

if reward > 100 and g.trust > 1.8:
    g.elite = True
    g.trust = 2.0  # Lock at maximum
    g.mutation_rate = 0.0  # Freeze mutations

# Elites never decay
if g.elite:
    # No decay
    pass
else:
    g.trust -= 0.007
```

Result: Mediocre genomes became immortal tyrants

Failure Mode:

1. Early lucky genome got 2-3 food (120 reward)

2. Became elite at trust=2.0
3. **Now immortal** - can't be dethroned
4. Population stuck with mediocre elite
5. Performance plateaued at 15-20% success

Example from logs:

csv

```
episode  success  best_genome_id  best_genome_trust
1000     0.18     0               2.0
2000     0.17     0               2.0
3000     0.16     0               2.0
4000     0.15     0               2.0  # Same genome for 4000 episodes!
```

Genome 0 monopolized the population for 4,000 episodes with terrible performance.

Elite System v2: Challenge Mechanism

Design: Elites can be dethroned by sustained superior performance

python

```python
# Non-elites can challenge if they outperform
if not g.elite and g.trust > 1.8:
    challenger_avg = sum(g.recent_rewards) / len(g.recent_rewards)

    worst_elite = min(elites, key=lambda e: e.recent_avg)

    # Dethrone if challenger is 20% better
    if challenger_avg > worst_elite_avg * 1.2:
        worst_elite.elite = False
        g.elite = True
```

Result: Musical chairs with mediocre genomes

Why it failed:

1. Non-elites still decayed 0.007 per episode
2. They could never build trust to 1.8 to challenge
3. Even good performers decayed faster than they could accumulate trust
4. **Elite rotation happened randomly, not meritocratically**

Performance:

- Oscillated 10-25% success
- Different genomes became elite, but all mediocre
- No improvement over 5k episodes

Elite System v3: Reduced Non-Elite Decay

Design: Make it possible for challengers to build trust

python

```
# Non-elites only decay on poor performance
if not g.elite:
    if reward < 5:  # Only actual failures
        g.trust -= 0.007
```

Result: Performance improved to 25-35% range, but still unstable

Progress:

- Non-elites could now reach 1.8 trust
- Legitimate challenges occurred
- Some elite rotation based on merit

Remaining Issues:

- Still occasional collapses when elites were dethroned
- No smooth succession - losing an elite caused temporary chaos
- Populations became dominated by 1-2 genomes

Key Insight #2

> **Elite systems with binary states (elite vs non-elite) create instability. Transitions are too abrupt and don't preserve population diversity.**

Phase 3: Trust as Currency

Philosophical Shift

After elite system failures, we reconsidered the fundamental nature of trust:

Previous view: Trust is a simple accumulator that needs explicit protection (elites)

New view: Trust should be **valuable, hard to earn, and protective by its nature**

Design: Trust Tiers with Asymmetric Dynamics

python

```
# HARDER to gain trust at higher levels
trust_gain = self.lr * reward

if g.trust > 1.8:
    trust_gain *= 0.3  # Crawl speed at top
elif g.trust > 1.5:
    trust_gain *= 0.5  # Half speed in veteran range

g.trust += trust_gain

# SLOWER decay for higher trust
if g.trust > 1.8:
    decay_rate = 0.0005  # Very slow
elif g.trust > 1.5:
    decay_rate = 0.002   # Slow
elif g.trust > 1.0:
    decay_rate = 0.005   # Moderate
else:
    decay_rate = 0.01    # Fast for low performers
```

Natural Trust Tiers Emerged:

- **0.0-1.0:** Proving ground (high churn, fast decay, easy to gain)
- **1.0-1.5:** Competent (moderate decay, slower gain)
- **1.5-1.8:** Veteran (slow decay, hard to reach)
- **1.8-2.0:** Elite (very slow decay, very hard to reach)

Results: First Stable Learning

Performance:

csv

```
episode  success  avg_reward  best_trust
1000     0.19     2.96        1.29
2000     0.25     8.41        1.17
3000     0.34     14.82       1.76
4000     0.39     15.70       1.78  # Peak
5000     0.31     11.03       1.23
6000     0.27     10.84       1.88
```

Breakthrough: First time maintaining 25-35% for 3,000+ episodes!

Why it worked:

- High trust protected good genomes naturally
- No arbitrary elite thresholds
- Smooth transitions - genomes could rise and fall gradually
- Trust became **scarce** at high levels (rarely saw 1.9+)

Remaining Issue: Still eventual collapse (39% → 27% over 2k episodes)

The Trust Ceiling Problem

Observation: Genomes with high trust (1.7+) and -4.83 avg reward

Example:


csv

```
episode  success  avg_reward  best_genome_id  best_genome_trust
1200     0.07     -4.83       10              2.0  # Trust 2.0 with 7% success!
```

Root cause: Trust reflected **historical performance, not current capability**

A genome could:

1. Have great episodes early (build to trust 2.0)
2. Stop performing well (population evolved past it)
3. Rarely get selected anymore
4. **Keep high trust indefinitely** (no decay when not used)

Fix: Recent Performance Cap


python

```
# Cap trust based on recent performance (last 30 episodes)
if len(g.recent_rewards) >= 30:
    recent_avg = sum(g.recent_rewards[-30:]) / 30
    max_trust = 0.5 + (recent_avg / 20)  # Maps reward to trust ceiling
    g.trust = min(g.trust, max(0.5, min(2.0, max_trust)))
```

Effect: Trust now reflected **current** capability, not ancient glory

Key Insight #3

> **Trust must reflect recent performance, not lifetime achievement. In continuous learning, "what have you done lately?" matters more than "what did you do once?"**

Phase 4: Relative Performance Discovery

The Collapse Pattern

Even with trust tiers and recent performance caps, we observed a repeating pattern:

csv

```
episode  success  avg_reward
1500     0.35     14.88     # Learning
1800     0.26     6.86      # Peak holding
2500     0.16     -0.16     # Decline begins
3000     0.06     -8.65     # Collapse
4000     0.05     -8.10     # Bottom
```

The cycle:

1. Population learns → performance climbs
2. High performers get protected by trust tiers
3. Population plateaus
4. Eventually, **all genomes decay together**
5. No one left who knows the good strategies
6. Catastrophic forgetting

Critical Realization

The problem wasn't individual genome decay - it was **context-blind protection**.

With trust tiers, we protected genomes above absolute thresholds:

- Trust > 1.8 → very slow decay

- Trust > 1.5 → slow decay

But this protected genomes even during population-wide collapse.

When the whole population declined, we were protecting "best of the worst" - genomes that had high trust but were no longer actually performing well relative to what the population had achieved before.

Relative Performance Design

New approach: Decay based on **population percentile rank**, not absolute trust

python

```
# Calculate genome's rank in current population
all_trusts = [gg.trust for gg in self.genomes]
sorted_trusts = sorted(all_trusts, reverse=True)
g_rank = sorted_trusts.index(g.trust) / len(sorted_trusts)  # 0=best, 1=worst

# Decay based on rank
if g_rank < 0.1:  # Top 10%
    decay = 0.00001  # Minimal
elif g_rank < 0.3:  # Top 30%
    decay = 0.002
else:
    decay = 0.007  # Aggressive
```

Result: Better, but still collapsed

Why: We were protecting "top 10% of declining population"

Context-Aware Protection

Final piece: Only protect high rankers if population is improving

python

```
# Check if population is improving
recent_pop_avg = sum(all_recent_rewards) / len(all_recent_rewards)
is_improving = reward > recent_pop_avg

# Only protect top performers during growth
if g_rank < 0.1 and g_recent_avg > 15:  # Top 10% AND quality gate
    if is_improving:
        decay = 0.00001  # Protect leaders during growth
    else:
        decay = 0.001    # Faster decay during collapse (find new strategies)
```

Breakthrough insight: During collapse, we need MORE exploration, not more protection

Results: Extended Stability

csv

```
episode  success  avg_reward  pattern
10000    0.30     10.65       # Baseline established
11000    0.31     9.91        # Holding
12000    0.31     11.87       # Holding
13000    0.27     8.10        # Small dip
14000    0.31     11.32       # Recovery!
15000    0.28     9.92        # Holding
16000    0.31     13.61       # Holding
17000    0.29     13.56       # Holding
18000    0.45     26.22       # BREAKTHROUGH!
19000    0.44     24.32       # Sustained!
20000    0.37     17.56       # Minor decline
```

First time holding 25-40% for 10,000 episodes!

Peak at episode 18k: **45% success rate, 26.22 avg reward**

Key Insight #4

> **Retention must depend on relative competence, not absolute thresholds. Protect the best performers, but only when they represent genuine quality, and only when the population is improving.**

Phase 5: The Breakthrough

The 10-Food Episode

After implementing relative, context-aware trust protection, we achieved the defining moment:

```
episode  success  eaten  total_reward  steps  best_genome_trust
4293     1        10     582.15        141    2.0
4394     1        8      491.03        156    2.0
3994     1        8      476.25        121    2.0
```

Episode 4293: 10 food eaten, 582.15 reward, 141 steps, trust 2.0

Significance:

- This was with **full self-collision detection enabled**
- Snake grew to length 10 (initial length 2)
- Navigated for 141 steps avoiding walls and its own body
- Achieved trust 2.0 - maximum protection
- Strategy was preserved (not a lucky fluke)

Context: Previous best with self-collision was 8 food. Without self-collision, we'd seen 12 food but that was an easier problem.

What Made It Possible

1. Fast Initial Learning The system consistently reached 40-50% success in 1,000-2,000 episodes:

csv

```
episode  success
100      0.17
500      0.18
1000     0.16
1500     0.35   # Rapid climb
2000     0.19
```

2. Trust Protection During Learning High performers naturally accumulated trust:

csv

```
episode  best_genome  trust
1500     14          1.17
1600     20          1.75   # Building trust
1700     22          2.0    # Hit maximum
1800     7           1.99   # Sustained
```

3. Strategy Preservation Once a genome achieved 10-food capability at trust 2.0, the ultra-low decay rate (0.00001) meant it could survive thousands of episodes:

- Decay per episode: 0.00001
- Decay over 1000 episodes: 0.01 (almost nothing)
- Trust floor for top 10%: 1.99+ indefinitely

Multiple High-Quality Episodes

It wasn't a single lucky run - the system produced multiple strong episodes:

```
7-8 food: Common (dozens of episodes)
5-6 food: Very common (hundreds of episodes)
10 food: Rare but achieved (proof of capability)
```

This demonstrated learned competence, not random exploration.

Visualization

At 400 FPS in the visualizer, we could watch the snake:

- Navigate purposefully toward food
- Avoid walls with planning (not reactive)
- Snake around its own body
- Make multi-step plans to reach food

Emergent strategy observed: The snake learned to favor open space - when growing long, it would position itself in the center of the grid rather than hugging walls.

Phase 6: Long-term Stability Challenge

The Collapse After Breakthrough

Despite achieving 10-food episodes, the system couldn't maintain peak performance indefinitely.

50k Episode Run Results:

csv

episode	success	best_trust	phase
1-10k	0.30	1.35	Learning
10-20k	0.45	2.0	BREAKTHROUGH
20-35k	0.14	0.68	Slow collapse
35-50k	0.10	0.44	Struggling

Pattern:

- Episodes 18k-19k: Peak at 45% success
- Episodes 20k-28k: Gradual decline (45% → 27%)
- Episodes 28k-35k: Accelerating collapse (27% → 14%)
- Episodes 35k-50k: Failed to recover

Trust trajectory:

- Episode 18k: best_trust = 1.99
- Episode 28k: best_trust = 1.81
- Episode 35k: best_trust = 0.97
- Episode 45k: best_trust = 0.54

Why It Failed Long-Term

Problem 1: Population-wide decline

Even with relative protection, when the ENTIRE population started declining:

- Top 10% was "best of declining population"
- Quality gate (recent_avg > 15) became hard to meet
- Protection was withdrawn from previous elites
- No genomes left that knew the 45% strategy

Problem 2: Insufficient mutation during plateau

Mutation triggered on `reward > median * 1.5`:

- During peak performance (median ~20), need reward >30
- Only ~10% of episodes triggered mutation
- Not enough exploration to escape local optima or rediscover lost strategies

Problem 3: Trust erosion during exploration

When trying new strategies (exploration):

- Genomes took temporary performance hits
- Trust decayed during experimentation
- Risk-averse: better to exploit known mediocre strategy than explore and lose trust

Attempted Fixes

Fix 1: Lower mutation threshold

python

```
if reward > median * 1.2:  # Was 1.5
    # Spawn mutation
```

Result: More mutations, but quality didn't improve

Fix 2: Protected mutation from decay

python

```
if attempting_exploration:
    # Don't decay during exploration window
```

Result: Hard to detect "exploration" vs "failure"

Fix 3: Absolute quality floor + relative rank

python

```
if g_rank < 0.1 and g_recent_avg > 15:  # AND condition
    # Both top ranked AND meets minimum quality
```

Result: Helped extend plateau, but didn't prevent eventual collapse

The Stability-Exploration Tradeoff

We discovered a fundamental tension:

Too much protection:

- Preserves strategies
- Prevents adaptation
- Can't escape local optima
- Stagnates then collapses when environment shifts

Too little protection:

- Allows exploration

- Loses breakthrough strategies
- Catastrophic forgetting
- Can't build on previous learning

The goldilocks zone is configuration-dependent and we haven't found it yet.

Key Learnings

1. Trust Dynamics Are Everything

Trust is the entire mechanism for continuous learning in OLA. Get it wrong and the system either:

- Forgets everything (too much decay)
- Stagnates forever (too little decay)
- Oscillates wildly (inconsistent decay)

The right balance requires:

- Decay proportional to relative performance, not absolute
- Context awareness (is population improving or declining?)
- Asymmetric dynamics (hard to gain at top, easy at bottom)
- Recent performance weighting (what have you done lately?)

2. Fixed Thresholds Are Death

Any hardcoded threshold in a continuous learning system becomes wrong:

- Reward thresholds (population improves, threshold becomes too easy)
- Trust thresholds (arbitrary, not adaptive)
- Time thresholds (environment changes)

Everything must be relative to current population state.

3. Elite Systems Need Careful Design

Binary elite/non-elite states create:

- Abrupt transitions (destabilizing)
- Monopolies (one genome dominates)
- Musical chairs (random rotation, not merit)

Better: Continuous gradients with natural stratification

4. Speed Enables Discovery

Running at **170-270 episodes/second** was crucial:

- Test configurations in minutes, not hours
- Rapid iteration on trust dynamics
- Observe long-term patterns (50k episodes in 3-5 minutes)

Without this speed, we couldn't have explored the parameter space effectively.

5. Breakthrough Discovery vs Retention Are Different

OLA proved very capable at:

- ✅ Fast learning (0-40% in 2k episodes)
- ✅ Discovering strategies (10-food episodes)
- ✅ Short-term retention (holding for 5-10k episodes)

But struggles with:

- ❌ Long-term stability (>20k episodes)
- ❌ Recovery from collapse
- ❌ Continuous improvement past initial plateau

These require different mechanisms.

6. The Population vs Individual Tradeoff

Protecting individuals (high-trust genomes) vs population diversity:

- Strong individual protection → less exploration → stagnation
- Weak individual protection → more exploration → forgetting
- **Need both:** protect best AND maintain diversity

Current system doesn't balance this well yet.

7. Relative Performance Needs Multiple Timescales

Current system uses:

- Recent rewards (last 50 episodes) for genome decisions
- Population median for decay decisions
- Single timescale for everything

Better approach might need:

- Short-term: immediate performance (last 10 episodes)
- Medium-term: trend (last 100 episodes)
- Long-term: baseline capability (best performance ever)

8. Context Matters More Than Absolute Performance

A genome with:

- Trust 1.8
- Recent average reward 25
- Top 5% of population

Should be treated differently if:

- **Population improving:** Protect strongly (leading the way)
- **Population declining:** Protect weakly (not good enough)

Current implementation does this, but parameters aren't tuned.

9. Catastrophic Forgetting Has Multiple Causes

We identified at least three distinct failure modes:

Type 1: Decay-induced (solved)

- Unconditional decay > learning rate
- Elite strategies decay while idle
- Solution: Conditional, rank-based decay

Type 2: Threshold-induced (solved)

- Fixed thresholds become obsolete
- Protection applies to wrong genomes
- Solution: Relative, context-aware thresholds

Type 3: Population-wide collapse (unsolved)

- All genomes decline together
- Even top 10% loses capability
- Protection fails because there's nothing good left to protect
- **This is the current blocker**

10. Meta-Learning Is Necessary

The fact that we need a parameter sweep proves:

- Trust dynamics are hypersensitive to configuration
- No single "obvious" configuration
- Need systematic search over parameter space

The system has the right structure, but wrong parameters.

What Worked

✅ Trust-Based Selection

Using trust-weighted sampling to select genomes:

python

```
def _select_genome_index(self) -> int:
    weights = [g.trust for g in self.genomes]
    return weighted_random_choice(weights)
```

Why it worked:

- Naturally favors high performers
- Maintains exploration (low-trust genomes still get chances)

- Smooth, continuous selection pressure
- No hard cutoffs

✅ Asymmetric Trust Dynamics

Making trust harder to gain at higher levels:

python

```
if g.trust > 1.8:
    trust_gain *= 0.3  # Crawl at top
```

Why it worked:

- Trust at 1.8+ became rare and valuable
- Created natural elite tier without explicit system
- Smooth stratification of population
- High trust = proven over many episodes

✅ Recent Performance Weighting

Deque with 50-episode rolling window:

python

```
g.recent_rewards = deque(maxlen=50)
g.recent_rewards.append(reward)
```

Why it worked:

- Genomes adapt to current environment
- Old performance doesn't dominate forever
- Enables detection of declining performers
- Memory efficient (capped size)

✅ Relative Rank-Based Decay

Decay based on percentile, not absolute values:

python

```
g_rank = sorted_trusts.index(g.trust) / len(sorted_trusts)
if g_rank < 0.1:
    decay = 0.00001  # Very slow for top 10%
```

Why it worked:

- Adapts to population capability
- Protects leaders regardless of absolute trust value
- Maintains competitive pressure throughout training
- No obsolete thresholds

✅ Quality Gate + Rank

Requiring both high rank AND minimum performance:


python

```
if g_rank < 0.1 and g_recent_avg > 15:
    # Protect
```

Why it worked:

- Prevents protecting "best of bad population"
- Ensures protection goes to genuinely good performers
- Combination is more robust than either alone
- Absolute floor prevents decay during population-wide collapse (somewhat)

✅ Mutation on Good Performance

Cloning successful genomes:


python

```
if reward > median * 1.5:
    weakest.copy_from(successful_genome)
    weakest.mutate()
```

Why it worked:

- Explores vicinity of good strategies
- Spreads successful patterns through population
- Maintains diversity (mutations, not perfect copies)
- Automatic - no manual intervention needed

✅ Two-Layer MLP Architecture

Simple 25 → 32 → 4 network:

Why it worked:

- Fast forward pass (critical at 170+ eps/sec)
- Sufficient capacity for Snake
- Easy to mutate (few parameters)
- No vanishing gradients (ReLU + no backprop anyway)

✅ Distance-Based Reward Shaping

Small rewards for moving toward food:

python

```
reward += 0.1 * (old_dist - new_dist)  # ±0.1 for distance change
```

Why it worked:

- Provides learning signal before first food
- Guides exploration toward productive behavior
- Small enough not to dominate food reward (+10)
- Enables bootstrapping from random policy

✅ Self-Collision Detection

Adding body-avoidance requirement:

Why it worked:

- Made the task realistic (full Snake game)
- Forced learning of spatial planning
- Prevented trivial solutions
- 10-food achievement more meaningful with this constraint

✅ Minimal, Fast Implementation

CPU-only, pure Python, no frameworks:

Why it worked:

- 170-270 episodes/second (critical for rapid iteration)
- Easy to modify (no framework constraints)
- Transparent (can debug every line)
- Reproducible (no GPU variance)

What Didn't Work

❌ Fixed Decay Rates

Any constant decay value:


python

```
g.trust -= 0.007  # Same for everyone, always
```

Why it failed:

- Too aggressive for high performers
- Too lenient for low performers
- Doesn't adapt to population state
- Caused catastrophic forgetting

❌ Fixed Elite Thresholds

Hardcoded promotion criteria:


python

```
if reward > 100 and trust > 1.8:
    g.elite = True
```

Why it failed:

- Threshold 100 arbitrary
- Became too easy (early) or too hard (late)
- Locked in mediocre genomes
- No adaptation to population improvement

❌ Binary Elite States

Elite vs non-elite as boolean:


python

```
g.elite = True  # Now immune to decay
```

Why it failed:

- All-or-nothing protection
- Abrupt transitions destabilized population
- Elite monopolies (one genome dominated)
- No smooth succession

❌ Immortal Elites Without Challenge

Permanent elite status:


python

```python
if g.elite:
    # Never decay, never lose status
    pass
```

Why it failed:

- Bad genomes became permanent elites
- No way to remove underperformers
- System stuck with first lucky genome
- Performance plateaued at 15-20%

❌ Absolute Performance Thresholds

Decay based on fixed reward values:


python

```python
if reward < 20:  # Hardcoded
    g.trust -= 0.007
```

Why it failed:

- Threshold became obsolete as population improved
- Early: too strict (everyone decayed)
- Late: too lenient (bad genomes protected)
- Not adaptive to population evolution

❌ Trust Based on Lifetime Performance

Trust accumulation without recency weighting:


python

```
g.trust += reward  # Forever accumulating
```

Why it failed:

- Ancient performance kept trust high
- Genomes that stopped performing kept protection
- Trust didn't reflect current capability
- "Retired" genomes cluttered top ranks

❌ Uniform Decay Application

Decaying selected AND idle genomes the same:


python

```
for g in all_genomes:
    g.trust -= decay  # Everyone decays
```

Why it failed:

- Elites decayed while waiting to be selected
- Punished genomes for not being chosen
- Good strategies lost during idle time
- Selection lottery, not meritocracy

❌ Challenge System Without Trust-Building

Elite challenge when non-elites couldn't reach threshold:


python

```
# Can challenge if trust > 1.8
# But non-elites decay before reaching 1.8
```

Why it failed:

- Challengers decayed before qualifying
- No legitimate challenges occurred
- Elite rotation became random
- Musical chairs with no improvement

❌ Context-Blind Protection

Protecting high-trust genomes regardless of population state:


python

```python
if g.trust > 1.5:
    decay = 0.0001  # Slow decay always
```

Why it failed:

- Protected "best of declining population"
- During collapse, needed more exploration not less
- Prevented system from finding new strategies
- Long-term stability impossible

❌ Insufficient Mutation Rate

Only mutating on very high rewards:


python

```python
if reward > median * 2.0:  # Rare
    mutate()
```

Why it failed:

- Too few mutations to explore
- Couldn't escape local optima
- Couldn't rediscover lost strategies
- Stagnation after initial learning

❌ Single Timescale for Everything

Using same rolling window for all decisions:


python

```python
recent_rewards = deque(maxlen=50)  # One size fits all
```

Why it failed:

- 50 episodes too short for long-term trends
- 50 episodes too long for immediate feedback
- Needed multiple timescales
- Couldn't detect different types of patterns

❌ No Population Diversity Mechanism

Only mutation for diversity:

Why it failed:

- Top genomes too similar (all clones + mutations)
- Lost orthogonal strategies
- Population collapsed to single approach
- Couldn't maintain multiple strategies simultaneously

❌ Ignoring Population-Wide Trends

Only looking at individual genome performance:

Why it failed:

- Couldn't detect population-wide collapse
- Didn't know when to increase exploration
- Protection failed when everyone was declining
- No meta-level adaptation

Current Best Configuration

Based on the longest stable run (18k-20k episodes at 40-45%), here's the current best configuration:

Core Parameters

python

```
# OLA initialization
num_genomes = 16
learning_rate = 0.01  # Trust gain rate
state_dim = 25  # 5x5 observation window
num_actions = 4

# Trust dynamics
trust_min = 0.0
trust_max = 2.0
```

Trust Gain (Asymmetric)

python

```
trust_gain = lr * reward * recency_weight

# Diminishing returns at high trust
if g.trust > 1.8:
    trust_gain *= 0.3  # 70% reduction
elif g.trust > 1.5:
    trust_gain *= 0.5  # 50% reduction

g.trust += trust_gain
```

Trust Decay (Relative + Conditional)

python

```python
# Calculate population context
all_trusts = [gg.trust for gg in genomes]
sorted_trusts = sorted(all_trusts, reverse=True)
g_rank = sorted_trusts.index(g.trust) / len(sorted_trusts)

# Recent performance
g_recent_avg = sum(g.recent_rewards) / len(g.recent_rewards)
recent_pop_avg = sum(all_recent_rewards) / len(all_recent_rewards)
is_improving = reward > recent_pop_avg

# Tiered decay based on rank and quality
if g_rank < 0.1 and g_recent_avg > 15:  # Top 10% + quality gate
    if is_improving:
        decay = 0.00001  # Nearly immortal during growth
    else:
        decay = 0.001    # Faster during population decline
elif g_rank < 0.3:  # Top 30%
    decay = 0.002
else:  # Bottom 70%
    decay = 0.007

# Apply decay only if below median performance
if reward < median:
    g.trust = max(0.0, g.trust - decay)
```

Mutation

python

```python
    # Trigger on above-average performance
    if reward > median * 1.5:
        weakest = min(genomes, key=lambda gg: gg.trust)

        # Clone successful genome
        weakest.w1 = copy(g.w1)
        weakest.b1 = copy(g.b1)
        weakest.w2 = copy(g.w2)
        weakest.b2 = copy(g.b2)
        weakest.trust = g.trust * 0.5  # Start with half trust

        # Mutate the clone
        weakest.mutate()  # mutation_rate = 0.05
```

Reward Structure

python

```python
    # Snake game rewards
    FOOD_REWARD = 10.0
    WALL_DEATH = -5.0
    DISTANCE_SHAPING = ±0.1  # per step
```

Performance Tracking

python

```python
    # Rolling windows
    recent_rewards = deque(maxlen=50)  # Individual genome
    success_window = 100  # For logging success rate
```

Known Limitations

This configuration:

- ✅ Learns reliably to 30-40% in 5k episodes
- ✅ Can achieve 45% peaks with 10-food episodes
- ✅ Holds performance for 5-10k episodes

- ❌ Eventually collapses after 20-30k episodes
- ❌ Doesn't recover well from collapse
- ❌ Parameters likely not optimal (need sweep)

What Needs Tuning

Based on observations, these parameters need optimization:

High Priority:

1. `quality_threshold` (currently 15) - balance between protection and exploration
2. `elite_rank` (currently 0.1) - how many to protect
3. `top_decay_multiplier` (currently 0.00001) - how strongly to protect
4. `mutation_threshold` (currently median * 1.5) - exploration rate

Medium Priority: 5. `good_rank` (currently 0.3) - moderate protection tier 6. `recent_rewards window` (currently 50) - timescale for performance 7. `recency_weight` - how much to favor recent vs old performance

Low Priority: 8. Learning rate (currently 0.01) - seems okay 9. Population size (currently 16) - seems okay 10. Network architecture (25→32→4) - seems okay

---

Next Steps: Systematic Parameter Sweep

Motivation

Manual tuning has revealed:

1. The system CAN learn and hold (proof of concept ✓)
2. Current parameters aren't optimal (collapses eventually)
3. Parameter space is large and non-obvious
4. Interactions between parameters matter

Solution: Systematic parameter sweep to find optimal configuration

Sweep Design

Parameter Grid:


python

```
sweep_params = {
    'base_decay': [0.002, 0.005, 0.007, 0.01],
    'top_10_decay_mult': [0.00001, 0.0001, 0.001],
    'top_30_decay_mult': [0.001, 0.002, 0.005],
    'quality_threshold': [10, 15, 20],
    'elite_rank': [0.05, 0.10, 0.15],
    'good_rank': [0.25, 0.30, 0.40],
}

# Total: 4 × 3 × 3 × 3 × 3 × 3 = 972 configurations
```

Metrics to Track:

- `peak_success_rate`: Maximum achieved
- `stability_duration`: Episodes maintaining >30% success
- `collapsed`: Boolean, did it collapse >20% from peak
- `episodes_to_30pct`: Learning speed
- `final_success_rate`: Last 1000 episodes
- `max_food_eaten`: Best single episode

Execution Plan

Phase 1: Smoke Test (Critical)

bash

```
python ola_parameter_sweep.py --smoke-test
```

Tests 3 random configurations for 1,000 episodes each (~2-3 minutes total)

Purpose:

- Verify all file saving works correctly
- Validate metrics make sense
- Confirm configuration parameters are actually used
- Check output format is correct

Success criteria:

- Creates timestamped output folder
- Saves JSON summaries with all config params
- Saves CSV history files
- Creates aggregate all_results.csv
- Prints progress and summary
- Completes in <5 minutes

If smoke test fails → Fix before proceeding to full sweep

Phase 2: Full Sweep

bash

```
python ola_parameter_sweep.py
```

- 972 configurations × 20,000 episodes each
- ~2-3 minutes per config
- **Total time: ~30-50 hours** (run overnight × 2 nights)
- Saves incrementally (can stop/resume)
- Tracks progress with ETA

Output:

```
sweep_results_20250115_143022/
├── config_001_summary.json
├── config_001_history.csv
├── config_002_summary.json
├── config_002_history.csv
├── ... (1944 files total)
├── all_results.csv          # Aggregate analysis
├── best_configs.json        # Top 10 by stability
└── sweep_metadata.json      # Sweep configuration
```

Analysis Plan

After sweep completes:

1. Identify Best Configurations

- Sort by `stability_duration` (primary)
- Filter: `peak_success_rate > 0.35`
- Check: `collapsed == False`

2. Analyze Parameter Patterns

- Which `quality_threshold` values work best?
- Optimal `elite_rank` percentage?
- Relationship between decay rates and stability?
- Parameter interactions (e.g., high protection + high quality threshold)

3. Validate Top 3 Configurations

- Re-run top 3 configs for 50k episodes

- Verify stability holds long-term
- Test with different random seeds
- Ensure reproducibility

4. Document Findings

- Optimal parameter set
- Parameter sensitivity analysis
- Failure mode analysis (why did bad configs fail?)
- Recommendations for future work

Expected Outcomes

Best case:

- Find configuration that holds 40%+ for 50k+ episodes
- Understand parameter sensitivities
- Prove stable continuous learning is possible
- Publishable results

Realistic case:

- Find configuration that holds 35-40% for 20-30k episodes
- Identify which parameters matter most
- Understand remaining failure modes
- Clear direction for next improvements

Worst case:

- All configurations eventually collapse
- Learn that additional mechanisms needed (not just parameter tuning)
- Understand limits of current approach
- Guide architectural changes

Timeline

- **Day 1:** Run smoke test (30 min), validate, start full sweep
- **Day 2-3:** Full sweep runs overnight (30-50 hours)
- **Day 4:** Analysis of results (4-8 hours)
- **Day 5:** Validation runs with top configs (4-8 hours)
- **Day 6:** Documentation and next steps

Total: 1 week from sweep start to validated optimal configuration

---

Conclusion

This journey from catastrophic forgetting to breakthrough discovery demonstrates that:

1. **Trust-based evolutionary learning can work -** we proved it with 10-food episodes and 45% success rates
2. **The devil is in the decay dynamics -** every failed approach taught us something about what doesn't work
3. **Continuous learning requires continuous adaptation -** fixed thresholds, absolute measures, and static protection all fail
4. **Speed enables discovery -** 170+ episodes/second lets us iterate rapidly and observe long-term patterns
5. **We're close but not there yet -** the system can learn and hold for 10k episodes, but not 50k+

The path forward is clear: systematic parameter sweep to find the configuration that balances protection with exploration, stability with adaptation, and retention with improvement.

The breakthrough moment - watching a 10-food episode unfold, with the snake navigating its own body and planning multi-step paths - proved that emergent intelligence is possible with pure evolution and trust-based selection.

Now we need to find the parameters that let that intelligence persist.

Appendix: Key Data Points

Best Episodes Achieved

```
Episode 4293: 10 food, 582.15 reward, 141 steps, trust 2.0
Episode 4394: 8 food, 491.03 reward, 156 steps, trust 2.0
Episode 3994: 8 food, 476.25 reward, 121 steps, trust 2.0
```

Longest Stable Performance

```
Episodes 10,000-20,000 (run #2):
- Maintained 27-44% success
- Peak at episode 18,000: 45% success
- Held above 30% for ~8,000 episodes
- Average trust: 1.4-2.0
```

Fastest Learning

```
Episodes 0-2,000 (run #3):
- 0% → 50% in 1,627 episodes
- Trust climbed to 1.99
- Then collapsed (insufficient retention)
```

Performance Metrics Summary

Speed: 170-270 episodes/second
Best success rate: 45% (episode 18,000)
Best food eaten: 10 (episode 4,293)
Longest stability: ~10,000 episodes
Typical learning time: 2,000-5,000 episodes to 30%+

Document Version: 1.0
Last Updated: January 15, 2025
Status: Parameter sweep pending, core mechanisms validated
Next Milestone: Identify optimal configuration via systematic sweep