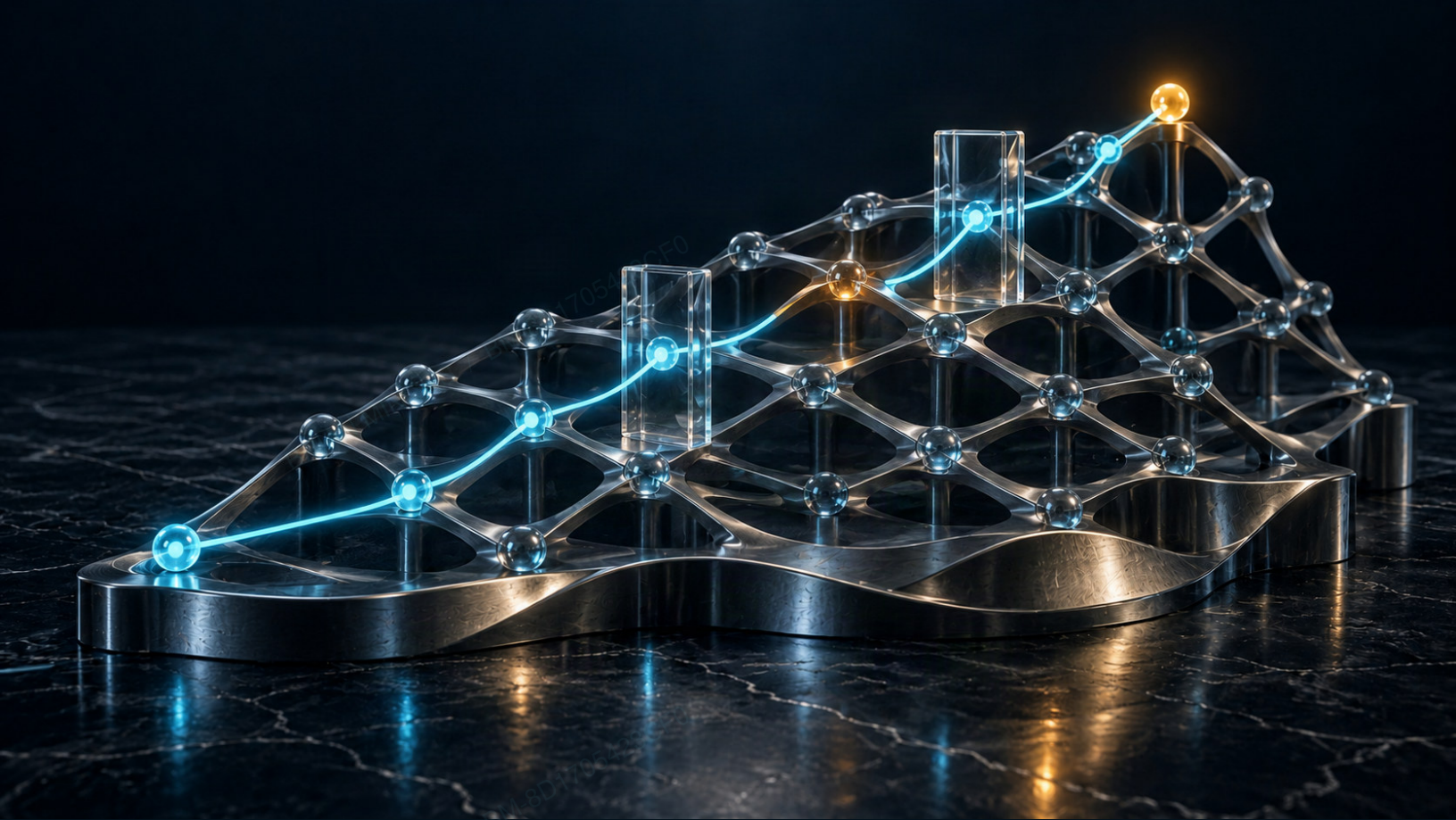




RESEARCH MANUSCRIPT | OCTOBER 2026

Brute-Force Search and Reinforcement Learning in 12-AX

Brian Mingus

An October 2026 rewrite of 2013/2014 work

Dense actor-critic learning and reward-guided policy search are compared in the 12-AX working-memory task, examining update costs, behavioral learning, and held-out performance.

Collection cover added October 3, 2026. Complete manuscript and appendices follow unchanged.

AI-generated conceptual artwork; separate from the scientific figures.

brianmingus.com

Brute-Force Search and Reinforcement Learning in 12-AX

Brian Mingus

Abstract

An adaptive policy can improve while retaining an expensive computational organization. We examine this distinction in the tabular working-memory model of Todd, Niv, and Cohen, using source analysis, numerical verification, and a random-search baseline for 12-AX. With K symbols, m memory slots, and A motor actions, the dense learner maintains $P = K^{m+1}(1 + A + 2m)$ parameters and performs $\Theta(P)$ learning work per transition. This exhaustive coordinate traversal grows exponentially with memory capacity. At an approximately 57.734-second budget, three random-search runs attain 97.950–99.667% held-out accuracy. In the seed-0 comparison, search achieves $d' = 5.108$ and the NumPy rewrite achieves $d' = 2.119$. An independent scalar-Python transcription completes fewer updates and performs worse under a different training protocol, although synchronized numerical tests confirm agreement of the learning equations. These results identify brute-force coordinate work and demonstrate the value of an explicit search baseline. They also distinguish equal runtime, equal learning rules, and equal experimental conditions: none alone establishes equivalence of learning and search.

1. Introduction

Learning curves describe improvements in behavior, but they do not by themselves explain the computation required to produce those improvements. A system may preserve successful behavior while continuing to spend substantial effort on uninformative trials, unused table entries, or repeated simulations. The relevant computational question is whether its learning rule reduces the effort needed to obtain a useful policy relative to an explicitly specified search procedure.

In *The Bitter Lesson*, Sutton (2019) identifies search and learning as general methods that benefit from increasing computation. This perspective motivates examining how each method spends its computational budget. The present study compares dense actor-critic learning with random

policy search in a finite working-memory task.

The finite-memory actor-critic model of Todd et al. (2008) makes this question concrete. The learner must discover both how to respond to observations and what to retain in working memory. Its state combines the current observation with the contents of independently gated memory slots. Tabulation gives a direct implementation, but the Cartesian representation becomes large as memory capacity increases. Eligibility traces further couple each observed transition to updates across the represented state space.

This paper studies brute-force computation at two distinct levels. At the implementation level, *exhaustive coordinate traversal* means visiting every entry of a tabular parameterization on every learning transition. At the algorithm level, the comparator is *reward-guided random search*: a candidate controller is proposed without gradients or a model of task dynamics, simulated, and retained when its scalar return is no worse. The comparator retains useful discoveries; its proposals remain uninformed by individual prediction errors or target labels.

Three results organize the analysis. First, the dense learning cost is derived as a function of alphabet size and memory capacity. Second, a direct numerical comparison separates agreement of the actor-critic equations from differences in reward, initialization, exploration, and execution speed. Third, held-out signal-detection measures characterize a random-search controller and two actor-critic implementations under their recorded computational budgets. The distinction between state-coordinate work and policy search is essential: the analysis does not identify dense table updates with enumeration of every possible policy.

2. Task and Controller Representation

2.1. The 12-AX task

Observations belong to

$$\mathcal{O} = \{1, 2, A, B, C, X, Y, Z\}.$$

The correct response is RIGHT when the current observation is X , the most recent digit is 1, and the most recent member of $\{A, B, C\}$ is A . The analogous 2-B-Y context also requires RIGHT. All other observations require LEFT. An observation alone is therefore insufficient to determine every

October, 2026 rewrite of 2013/2014 work.

response.

In the task specified by Todd et al. (2008, Section 3.1), a correct response to X or Y earns reward 1, any other correct response earns 0.25, and an error earns 0. The NumPy rewrite studied here uses a different reward convention: correct RIGHT earns 1, while every correct LEFT earns 0.25. The conventions differ on correct LEFT responses to X or Y . They have the same correct-response labels once context is established, but produce different learning signals.

Two stimulus protocols are used. The rewrite benchmark draws symbols independently and uniformly and organizes them into fixed-length episodes. The independent equation implementation uses structured sequences derived from the benchmark of O'Reilly & Frank (2006, Section 4.1): a digit is followed by one to four cue–probe pairs; each pair is either the context-appropriate target pair or a random cue and probe, with equal branch probability. Fair digits, uniform pair counts, and independent uniform distractor draws complete this specification. This is an explicit reconstruction of a cited protocol; Todd et al. do not provide the sampling details needed to identify their exact generator.

2.2. State and policy spaces

The internal state is $s_t = (o_t, w_t^1, \dots, w_t^m)$, where each memory element holds a symbol from the observation alphabet. A gate either maintains its memory or writes the current observation. The motor actor and all gates choose from the same pre-update state. The next observation and updated memories then define s_{t+1} .

For K symbols and m slots, the Cartesian table has $N = K^{m+1}$ states. The 12-AX implementation has $K = 8$, $m = 2$, and $N = 512$. One binary motor actor and two binary gates provide eight joint actions per state. The actor–critic factorizes its stochastic policy into these three distributions. Their deterministic limit can represent any assignment of one joint action to each row, which is the controller class searched by the comparator. That class contains 8^{512} controllers.

3. What the Dense Learner Computes

3.1. Learning equations

The learner maintains a critic V , motor preferences u , gate preferences v_j , and corresponding eligibility traces. Its temporal-difference error is

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t), \quad (1)$$

with $\gamma = 0.94$, $\alpha = 0.1$, and $\lambda = 0.75$ in the implementations compared here. For an actor with preferences θ and

policy π , Table 1 of Todd et al. (2008) specifies

$$e_\theta(s, a) \leftarrow \begin{cases} \mathbf{1}[a = a_t] - \pi(a | s_t), & s = s_t, \\ \gamma \lambda e_\theta(s, a), & s \neq s_t, \end{cases} \quad (2)$$

and the dense update $\theta \leftarrow \theta + \alpha \delta_t e_\theta$. The critic trace instead accumulates an increment at the current state:

$$e_V(s) \leftarrow \gamma \lambda e_V(s) + \mathbf{1}[s = s_t], \quad (3)$$

$$V(s) \leftarrow V(s) + \alpha \delta_t e_V(s). \quad (4)$$

These equations apply separately to the motor actor and each gate. Both implementations follow the printed actor-trace expression without adding an inverse-temperature multiplier.

3.2. Exhaustive coordinate traversal

Proposition 1 (Dense representation cost). *For a tabular critic, A motor actions, and m binary gates over $N = K^{m+1}$ states, define*

$$P = N(1 + A + 2m). \quad (5)$$

An implementation that explicitly updates every parameter and eligibility coordinate requires $\Theta(P)$ arithmetic work per transition and $\Theta(P)$ persistent table storage. Over T transitions, its learning work is $\Theta(TP)$ under unit-cost scalar arithmetic.

Proof. The critic has N values, the motor actor has AN preferences, and the gates have $2mN$ preferences. Their traces have identical dimensions, giving P parameters and P traces. Every transition writes every trace and parameter coordinate, providing an $\Omega(P)$ lower bound. Each coordinate requires a bounded number of scalar operations, providing an $O(P)$ upper bound. The two persistent collections require $2P$ scalars. $\square$

For 12-AX, $P = 3,584$ and parameter-plus-trace storage is 7,168 floating-point values, excluding temporaries and environment state. The scalar-Python implementation traverses these coordinates even when their eligibility values are zero. NumPy evaluates the rewrite's corresponding dense expressions in compiled array routines; this changes execution cost without changing the mathematical update.

For fixed $K > 1$ and A , Equation 5 grows exponentially with m . This result concerns the explicit dense representation. It is not a lower bound for all implementations of these equations, alternative memory representations, or reinforcement learning in general. In particular, the $\Theta(P)$ traversal does not evaluate the exponentially larger set of deterministic policies.

The exact counts in Table 1 distinguish arithmetic from elapsed time. The rewrite multiplies actor traces by λ and

Table 1. Elementwise work in the dense learning expressions of the two original source listings. Scalar setup, active-row edits, indexing, allocation, and policy sampling are excluded.

Operation	Initial version	Rewrite
Multiplications	$14N$	$20N$
Additions	$7N$	$7N$
Explicit copied entries	$7N$	$7N$

Algorithm 1 Random policy mutation with scalar-return retention

```

1: Draw every row of  $q$  uniformly from eight actions.
2:  $J \leftarrow \text{Score}(q, \mathcal{B})$ .
3: while the training budget remains do
4:   if a training-bank refresh is due then
5:     Draw  $\mathcal{B}$  anew;  $J \leftarrow \text{Score}(q, \mathcal{B})$ .
6:   end if
7:   Copy  $q$  to  $q'$ ; draw a row  $s$  and action  $a$  uniformly.
8:    $q'(s) \leftarrow a$ ;  $J' \leftarrow \text{Score}(q', \mathcal{B})$ .
9:   if  $J' \geq J$  then
10:     $(q, J) \leftarrow (q', J')$ .
11:   end if
12: end while

```

then γ in separate array expressions; the initial version caches their scalar product. Additional work in the first version comes from prepending observations to an unbounded history list. Front insertion alone shifts $(T^2 + 5T)/2$ list references over T transitions. With fixed alphabet and actor widths, total program work is $\Theta(TN + T^2)$. The rewrite replaces that list with last-occurrence counters, giving $\Theta(TN)$ work under the same model. These bookkeeping costs illustrate why similar runtimes need not imply similar search behavior.

4. Random Search over Memory Policies

The comparator stores a deterministic table $q(s) \in \{0, \dots, 7\}$. Three action bits encode a motor response and two memory gates. It initializes each row uniformly, selects a row uniformly for each proposal, and replaces its action with a uniform draw from all eight actions. Unchanged proposals are allowed and are still evaluated. The complete candidate is simulated on the current bank of episodes; the only feedback to the proposal-and-retention procedure is the summed reward. Improvements and ties replace the incumbent.

The reported variant uses eight episodes of 1,000 transitions. It retains the initial bank for 16,384 proposals, then refreshes before the next proposal and every 128 proposals thereafter. Each refresh regenerates the bank and re-evaluates the incumbent. Random generators for data and policy proposals are independent. The warmup and refresh schedule were

selected through exploratory pilots; the results are not a preregistered comparison.

The evaluator prepares observation offsets and reward lookup tables that depend only on the episode. This work is charged to training initialization or refresh. Every candidate’s memory updates and motor actions are nevertheless simulated from the beginning using ordinary Python loops. No candidate returns or policy-dependent trajectory segments are reused. The search receives no target labels, gradients, value estimates, or visitation-based proposal bias. Because candidates are retained by reward, the method is a stochastic hill-climber rather than independent sampling of complete policies or exhaustive policy enumeration.

For fixed memory capacity, scoring B episodes of length L costs $\Theta(BL)$ policy-dependent work per proposal. Copying the controller adds $\Theta(N)$. This cost structure differs from performing a dense actor–critic update at every observed transition. A search simulation step and an actor–critic learning step therefore measure different amounts of computation.

5. Verification and Measurement

5.1. Implementations and numerical agreement

The comparison includes a callable Python 3 port of the NumPy rewrite, the random-search comparator, and an independent dense transcription of the paper equations using Python lists and scalar loops. The port preserves the rewrite’s numerical update order and reward conventions. Its numerical arrays, random-generator states, actions, and counters agree with execution of the original source over 5,032 transitions across 13 qualification cases.

To test the learning equations directly, the independent transcription was initialized with the rewrite’s parameters and supplied with the same states, sampled actions, and rewards. Across 3,000 transitions from three seeds, 21,525,000 scalar comparisons gave a maximum absolute discrepancy of 2.11×10^{-15} . Critic values, critic traces, and temporal-difference errors were identical in these cases. This verifies agreement of the tested update computations independently of stimulus sampling and reward interpretation. It does not require independently sampled runs to follow the same trajectory.

5.2. Training configurations

Table 2 states the configurations rather than treating implementation names as a complete experimental specification. The rewrite cycles an eight-episode bank while retaining learned parameters and its action-generator state across episode resets. Random search begins from the same training bank but subsequently refreshes it. The indepen-

Table 2. Actor-critic settings in the common-budget comparison. The equation implementation uses the configuration that learned the structured task in the separate long-run study. The repeated-bank adapter is part of the present benchmark, not part of the original standalone program.

Setting	NumPy rewrite	Equation implementation
Numerical execution	Dense NumPy operations	Dense scalar Python loops
Softmax temperature	0.04	1.0
Initial gate preferences (update, maintain)	(0, 0.16)	(0, 0)
Initial maintenance probability	≈ 0.982	0.5
Initial critic value	4.68	$1/(1 - \gamma) \approx 16.67$
Correct LEFT reward on X/Y	0.25	1.0
Training observations	Uniform-symbol episode bank, cycled	Continuous structured sequences
Memory/history/trace resets	Every 1,000 transitions	None during training
Common settings	$N = 512$, two gates, $\alpha = 0.1$, $\gamma = 0.94$, $\lambda = 0.75$; zero initial motor preferences and traces	

dent equation implementation learns continuously on the structured protocol and uses the paper’s reward rule.

The paper does not give numerical values for temperature, initial preferences, or critic initialization. The settings in Table 2 are therefore experimental choices. The new critic value is an optimistic bound motivated by a maximum reward of one and discount 0.94. In particular, equal actor-critic equations do not imply equal initial action distributions: the rewrite’s gate bias yields $\Pr(\text{maintain}) = 1/(1 + e^{-4})$, whereas unbiased gates initially maintain with probability one half.

5.3. Budgets and timing

All numerical experiments reported here were performed in October 2026 using the recovered 2014 implementations and the new comparison programs. Measurements use an Intel Xeon W-11855M host, Windows 11, Python 3.12.14, and NumPy 2.3.5 where applicable. The rewrite endpoint contains 512,000 learning transitions, the cap of a preceding validation schedule. None of its assessed checkpoints met that schedule’s criterion of MCC at least 0.5 and both class recalls at least 0.8. Its cumulative elapsed training time, approximately 57.733 seconds, set the search budget; the scalar equation run uses a requested budget of 57.734 seconds. Training stops after a complete proposal or transition. Timing includes initialization and training work, with bank regeneration, preparation, and incumbent rescoring charged to search. Initial shared banks are supplied before timing; their preparation is included. Frozen evaluation, checkpoint serialization, and file output are excluded.

Elapsed time, process CPU time, and Windows process cycles are retained separately. Process cycles are an execution measurement, not a source-operation or instruction count. The rewrite and search timings are archived measurements, whereas the scalar-Python run was measured subsequently. This is a comparison of recorded endpoints at a common elapsed budget, not a simultaneous hardware-controlled trial

or a measurement of time to a common proficiency.

5.4. Held-out evaluation and signal detection

The common test contains 64 independent episodes of 1,000 transitions: 2,582 RIGHT targets and 61,418 non-targets. Each policy receives exactly the same observations and initial memories. The historical simulator’s history initialization is also shared. Actor-critic policies are read greedily, with ties assigned to the lower-index action; random search is already deterministic. The test is a distribution-transfer assessment for the equation implementation, which was trained on structured observations. No model updates occur during evaluation.

Let TP, FN, FP, and TN denote true positives, misses, false alarms, and correct rejections, with RIGHT designated as the signal response. Raw hit and false-alarm rates are $H = TP/(TP + FN)$ and $F = FP/(FP + TN)$. All non-target symbols contribute to the false-alarm denominator. To obtain finite signal-detection indices, the log-linear correction adds 0.5 to each contingency-table cell (Hautus, 1995):

$$\tilde{H} = \frac{TP + 0.5}{TP + FN + 1}, \quad \tilde{F} = \frac{FP + 0.5}{FP + TN + 1}, \quad (6)$$

$$d' = z(\tilde{H}) - z(\tilde{F}), \quad (7)$$

$$c = -\frac{1}{2}[z(\tilde{H}) + z(\tilde{F})]. \quad (8)$$

Here $z = \Phi^{-1}$. These are descriptive equal-variance signal-detection indices; positive c indicates a more conservative RIGHT response. Matthews correlation coefficient (MCC) is reported alongside accuracy. Uncertainty across training runs cannot be inferred by treating all temporally dependent observations as independent replicates.

5.5. Reproducibility

Code and data accompany this paper as `rl-whitepaper-reproducibility.zip`. The archive contains implementation sources, immutable run plans, stimulus banks,

Table 3. Common-budget, same-test endpoints, seed 0. Training protocols differ as specified in Table 2; only evaluation conditions and approximate elapsed budgets are shared. Accuracy, hit rate, and false-alarm rate are percentages. Signal-detection indices use the correction in Section 5.

Implementation	Wall (s)	CPU (s)	Accuracy	Hit	False alarm	d'	c	MCC
NumPy actor-critic rewrite	57.733	56.875	92.655	72.773	6.509	2.119	+0.454	0.451
Random policy search	57.734	57.453	98.614	99.845	1.438	5.108	-0.367	0.856
Scalar-Python actor-critic	57.735	57.453	67.928	79.861	32.574	1.288	-0.192	0.196

trained-model records, confusion counts, and numerical checks. Its evidence manifest maps the reported measurements to their source files and hashes. Original Python listings are reproduced without modification in Appendices A and B.

6. Results

6.1. Computation and generalization at a fixed budget

Table 3 reports the common-test outcomes. Random search achieves 98.614% accuracy, with four misses and 883 false alarms. The rewrite has 703 misses and 3,998 false alarms. The scalar-Python equation run has 520 misses and 20,006 false alarms. Since an always-LEFT controller attains 95.966% accuracy on this test, accuracy alone would conceal the rewrite’s substantial sensitivity and the failure of that trivial policy to detect any target. The random-search result exceeds the always-LEFT accuracy while maintaining a 99.845% hit rate.

The recorded workloads differ markedly. The rewrite completes 512,000 learning transitions; the scalar equation implementation completes 57,844, a ratio of 8.85. Search completes 54,506 proposals and 298 bank refreshes, totaling 438,440,000 scored transitions including initial and refresh evaluations. Its prepared scalar simulator makes these transitions much cheaper than dense actor-critic learning updates. Neither the transition-count ratio nor the equal elapsed budget is an algorithmic-equivalence result.

The scalar implementation’s poor common-test score does not contradict numerical agreement of the equations. Its run has fewer learning opportunities, different initial action probabilities, different rewards, and a different training distribution. On a separate 100,000-observation structured test, the same frozen model attains 77.739% accuracy, $H = 90.836\%$, $F = 26.232\%$, and $d' = 1.967$ with stochastic readout. Greedy readout on that same structured stream gives 80.043% accuracy. These evaluations separate transfer behavior from performance on the training distribution without selecting a different checkpoint.

6.2. Replication and longer training

The same search configuration was run with seeds 0, 1, and 2, each evaluated on a separate fresh bank of 64,000 tran-

Table 4. All three runs of the selected search configuration, each with an approximately 57.734-second training budget. Rates are percentages; d' uses corrected counts.

Seed	Accuracy	Hit	False alarm	d'	MCC
0	98.614	99.845	1.438	5.108	0.856
1	97.950	99.045	2.097	4.370	0.805
2	99.667	99.622	0.331	5.369	0.960

sitions. Table 4 reports every seed. Accuracy ranges from 97.950% to 99.667%, and MCC from 0.805 to 0.960. The three-way comparison remains a single seed-0 comparison: the additional search runs do not supply missing paired actor-critic replicates.

A separate 20-million-transition study tests whether the independent equations can learn the structured task under the optimistic initialization. All three seeds attain at least 99.604% accuracy on fresh 100,000-observation stochastic tests; the range is 99.604–99.984%. These runs use a separately qualified compiled diagnostic whose short trajectories agree numerically with the scalar Python implementation. They establish learning under that configuration, but their compiled timings are not Python timings and are excluded from Table 3.

The long-run configuration was selected after initialization and protocol diagnostics. Across 24 runs covering eight protocol-initialization combinations, four meet the stated joint criterion of accuracy at least 99%, target recall at least 95%, non-target recall at least 99%, and MCC at least 0.95. Three use the optimistic initialization in Table 2; one uses a lower positive critic initialization under the same structured protocol. All configurations and outcomes remain part of the experimental record. This is evidence of configuration sensitivity, not an unbiased estimate of a generally optimal setting.

7. Interpretation

7.1. Discovery, retention, and computational advantage

Reward-guided retention is sufficient for the comparator to develop a useful memory policy without a critic or eligibility traces. The result supplies a concrete alternative explanation for success on the benchmark: repeated policy

simulation and selective retention can produce high sensitivity. The present measurements do not establish that temporal-difference learning and this search process discover policies at the same rate, or that either dominates across task distributions.

7.2. What a timing comparison can establish

At a fixed budget, the natural outcome is the distribution of held-out performance. At a fixed proficiency, the outcome is time to attainment, with non-attaining runs retained as censored observations. Running every program for the same number of seconds enforces equality of the budget; it does not demonstrate equal time to learn.

The current common-test result establishes the performance of three specified endpoints. A comparison intended to attribute differences to the learning rule must additionally match training observations, rewards, initialization, reset semantics, and allowed information, and must report multiple independent training runs. Computation per transition should be separated from the number of transitions required for proficiency. Search also has repeated access to episode banks, a resource that must be stated when comparing it with a continuing online learner.

The core computational finding is more direct. Dense tabulation performs exhaustive coordinate work even while its policy improves, and that work scales exponentially with the number of memory slots. This form of brute-force computation is compatible with correct reinforcement-learning updates. It is a property of the implementation and representation, rather than a claim that every reinforcement-learning method enumerates policies or has the same cost as random search.

8. Conclusion

The 12-AX actor–critic and the random-search comparator use the same deterministic controller class but spend computation differently. The dense learner traverses its full parameterization on each transition; search repeatedly simulates candidate controllers and retains nondecreasing reward. Source analysis proves the dense representation’s cost, numerical checks confirm agreement of the actor–critic learning equations, and held-out measurements show that simple reward-guided search can achieve strong signal detection within the recorded budget. Separating these findings makes the computational claim precise: learning can coexist with brute-force coordinate work, while any stronger equivalence between learning and policy search requires a controlled comparison of discovery cost and attained performance.

Impact Statement

This work examines computational cost and experimental measurement in a synthetic working-memory task. Its practical contribution is a reproducible distinction between learning behavior, implementation overhead, and search baselines. No human-subject data are used in the experiments.

References

- Hautus, M. J. Corrections for extreme proportions and their biasing effects on estimated values of d' . *Behavior Research Methods, Instruments, & Computers*, 27(1):46–51, 1995. doi: 10.3758/BF03203619.
- O’Reilly, R. C. and Frank, M. J. Making working memory work: A computational model of learning in the prefrontal cortex and basal ganglia. *Neural Computation*, 18(2):283–328, 2006. doi: 10.1162/089976606775093909. URL <https://ski.clps.brown.edu/papers/OReillyFrank06.pdf>.
- Sutton, R. S. The bitter lesson. Incomplete Ideas (blog), March 13, 2019. URL <https://incompleteideas.net/IncIdeas/BitterLesson.html>.
- Todd, M. T., Niv, Y., and Cohen, J. D. Learning to use working memory in partially observable environments through dopaminergic reinforcement. In Koller, D., Schuurmans, D., Bengio, Y., and Bottou, L. (eds.), *Advances in Neural Information Processing Systems*, volume 21. Curran Associates, Inc., 2008. URL https://proceedings.neurips.cc/paper_files/paper/2008/file/98dce83da57b0395e163467c9dae521b-Paper.pdf.

A. Original Python Implementation

The following listing preserves the original implementation, including Python 2 syntax, history-list representation, and the 200,000-transition stopping rule. The computational counts in Table 1 refer to its dense expressions. The measured NumPy endpoint instead uses the rewrite in Appendix B through the benchmark adapter described in Section 5.

Source revision: [5f01ee704705379a1b06fa54d04db369fc294217](https://github.com/brm481/12-AX-RL/blob/5f01ee704705379a1b06fa54d04db369fc294217).

Git blob: [ea4577de17f5afcd60cf04e1075fff88236b707c](https://github.com/brm481/12-AX-RL/blob/5f01ee704705379a1b06fa54d04db369fc294217).

```

1  #!/bin/python
2  from sys import argv
3  from pprint import pprint as pp
4  from numpy.random import choice
5  import numpy as np
6  from sys import maxint
7  import cProfile
8
9  try:
10     temp = float(argv[1])
11     maint_pref = float(argv[2])
12     batch = argv[3]
13 except:
14     temp = .04
15     maint_pref = .16
16     batch = 0
17
18 f = open("log/temp_" + str(temp) + "_maint_pref_" + str(maint_pref) + "_batch_" + str(batch) + ".log",
19 ↪ "w")
20 f.write("trial\tmax_scr\t\t\t\n")
21 f.flush()
22 Os = ['1','2','A','B','C','X','Y','Z'] # states
23 Ss = [i+j+k for i in Os for j in Os for k in Os]
24
25 def softmax(w, t = temp):
26     e = np.exp(w / t)
27     return e / e.sum()
28
29 def inv(act_i):
30     return int(not bool(act_i))
31
32 def j(s_t):
33     try:
34         return h.index(s_t)
35     except:
36         return maxint
37
38 # action value diff should be ~.2
39 def avd():
40     print np.sum(np.array(u_prefs)[: ,1])/512.0 - np.sum(np.array(u_prefs)[: ,0])/512.0
41
42 h = [] # history
43 u_acts, v1_acts, v2_acts = ['L','R'], ['U','M'], ['U','M'] # actions
44
45 def init2d(val1, val2): return np.array([[val1, val2] for i in range(len(Ss))])
46 def init1d(val): return np.array([[val] for i in range(len(Ss))])
47
48 # initialize 2d arrays
49 u_prefs, v1_prefs, v2_prefs = init2d(0.,0.), init2d(0.,maint_pref), init2d(0.,maint_pref)
50 e_M, e_G1, e_G2 = init2d(0.,0.), init2d(0.,0.), init2d(0.,0.)
51 e_V, V = init1d(0.), init1d(4.68)
52
53 # initialize state
54 s_t = choice(Os) + choice(Os) + choice(Os)
55 s_t_i = Ss.index(s_t)
56 h.insert(0,s_t[2])
57 h.insert(0,s_t[1])
58 h.insert(0,s_t[0])
59 o = s_t[0]
60
61 # params
62 l = 0.75 # eligibility trace decay rate

```

```

63 g = 0.94 # discount rate
64 lg = 1*g
65 a = 0.1 # lrate
66 r = 0.0
67 wm = [s_t[1], s_t[2]]
68 t = 0
69
70 # performance
71 tp, fp, tn, fn = 0.0, 0.0, 0.0, 0.0
72 scr = 0.0 # successive correct responses
73 scrs = []
74 max_scr = 0.0
75 n_right = 0
76
77 while True:
78     ## Choose motor and gating actions for current state
79     # choose actions for this state
80     u_sm = softmax(u_prefs[s_t_i])
81     v1_sm = softmax(v1_prefs[s_t_i])
82     v2_sm = softmax(v2_prefs[s_t_i])
83
84     u = choice(u_acts, p = u_sm)
85     v1 = choice(v1_acts, p = v1_sm)
86     v2 = choice(v2_acts, p = v2_sm)
87
88     ## Update motor and gating action eligibility traces, e_M, e_G1, e_G2
89
90     # e_M = 1 * g * e_M, s != s_t
91     e_M *= lg
92     e_G1 *= lg
93     e_G2 *= lg
94
95     u_i, v1_i, v2_i = u_acts.index(u), v1_acts.index(v1), v2_acts.index(v2)
96     n_u_i, n_v1_i, n_v2_i = inv(u_i), inv(v1_i), inv(v2_i)
97
98     # e_M = 1 - Pr(a/s), s = s_t, a = a_t
99     e_M[s_t_i][u_i] = 1 - u_sm[u_i]
100     e_G1[s_t_i][v1_i] = 1 - v1_sm[v1_i]
101     e_G2[s_t_i][v2_i] = 1 - v2_sm[v2_i]
102
103     # e_M = - Pr(a/s), s = s_t, a != a_t
104     e_M[s_t_i][n_u_i] = -u_sm[n_u_i]
105     e_G1[s_t_i][n_v1_i] = -v1_sm[n_v1_i]
106     e_G2[s_t_i][n_v2_i] = -v2_sm[n_v2_i]
107
108     ## Get reward and next state
109     # get the indexes of each element in the queue
110     ONE, TWO, A, B, C, X, Y, Z = j('1'), j('2'), j('A'), j('B'), j('C'), j('X'), j('Y'), j('Z')
111     if (X == 0 and A < B and A < C and ONE < TWO) or (Y == 0 and B < A and B < C and TWO < ONE):
112         if u == 'R':
113             r = 1.0
114             tp += 1
115             scr += 1
116             n_right += 1
117         elif u == 'L':
118             r = 0.0
119             fn += 1
120             if scr > max_scr:
121                 scrs.append(scr)
122                 max_scr = scr
123                 f.write(str(t) + "\t" + str(max_scr) + "\t" + str(n_right) + "\n")
124                 f.flush()
125                 scr = 0
126                 n_right = 0
127     elif u == 'L':
128         r = 0.25
129         tn += 1
130         scr += 1
131     elif u == 'R':
132         r = 0.0
133         fp += 1
134         if scr > max_scr:
135             scrs.append(scr)
136             max_scr = scr

```

```

137         f.write(str(t) + "\t" + str(max_scr) + "\t" + str(n_right) + "\n")
138         f.flush()
139         scr = 0
140         n_right = 0
141
142         # Update internal state based on previous state, gating action and new observation
143         if v1 == 'U': wm[0] = o
144         if v2 == 'U': wm[1] = o
145
146         # New trial starts here from the environment's perspective
147
148         # randomly choose the next state and update history
149         o = choice(Os)
150         h.insert(0, o)
151
152         s_nxt = o + wm[0] + wm[1]
153         s_nxt_i = Ss.index(s_nxt)
154
155         # compute state-value prediction error based on critic's state-value approximation
156         d_t = r + g * V[s_nxt_i] - V[s_t_i]
157
158         e_V *= g*1
159         e_V[s_t_i] += 1
160         e_V_tmp = np.copy(e_V)
161         e_V_tmp *= a*d_t
162         V += e_V_tmp
163
164         e_M_tmp = np.copy(e_M)
165         e_M_tmp *= a*d_t
166         u_prefs += e_M_tmp
167
168         e_G1_tmp = np.copy(e_G1)
169         e_G1_tmp *= a*d_t
170         v1_prefs += e_G1_tmp
171
172         e_G2_tmp = np.copy(e_G2)
173         e_G2_tmp *= a*d_t
174         v2_prefs += e_G2_tmp
175
176         s_t, s_t_i = s_nxt, s_nxt_i
177
178         t += 1
179
180         # print t, max_scr, scr, s_nxt, u
181
182         if t % 200000 == 0:
183             f.close()
184             exit()
185
186         # if t % 1000 == 0:
187         #     tp, fp, tn, fn = 0.0, 0.0, 0.0, 0.0
188         # if r == 1.0:
189         #     try:
190         #         precision = tp / (tp + fp)
191         #         recall = tp / (tp + fn)
192         #         accuracy = (tp + tn) / (tp + tn + fp + fn)
193
194         #         print t, "precision: " + str(precision) + "\trecall: " + str(recall) + "\tfmeasure: " +
195         ↪ str(accuracy)
196         #     except:
197         #         pass

```

B. Original Python Rewrite

This listing preserves the rewrite's last-occurrence counters, initialization, reward rule, and 1,000-transition stopping rule. The callable port removes standalone logging and termination while retaining its numerical semantics. The benchmark adapter adds repeated episodes and transient-state resets; those additions are not present in this listing.

Source revision: [23158c64a65fd5800adc895258f4d536283ddfca](#).

Git blob: [2f4fffc232b01444a7f77e08310e58dd8bf6aaef4](#).

```

1  #!/bin/python
2  from sys import argv
3  from pprint import pprint as pp
4  from numpy.random import choice
5  from numpy import exp, sum, array, copy
6  import cProfile
7
8  try:
9      temp = float(argv[1])
10     maint_pref = float(argv[2])
11     batch = argv[3]
12 except:
13     temp = .04
14     maint_pref = .16
15     batch = 0
16
17 f = open("log/temp_" + str(temp) + "_maint_pref_" + str(maint_pref) + "_batch_" + str(batch) + ".log",
18 ↪      "w")
19 f.write("trial\tmax_scr\t\t\t\n")
20 f.flush()
21
22 Os = ['1', '2', 'A', 'B', 'C', 'X', 'Y', 'Z'] # states
23 Ss = [i+j+k for i in Os for j in Os for k in Os]
24
25 def sm(w, t = temp):
26     e = exp(w / t)
27     return e / sum(e)
28
29 # action value diff should be ~.2
30 def avd(): print sum(array(u_prefs)[: ,1])/512.0 - sum(array(u_prefs)[: ,0])/512.0
31 def init2d(vall, val2): return array([[vall, val2] for i in range(len(Ss))])
32 def init1d(val): return array([[val] for i in range(len(Ss))])
33
34 # initialize 2d arrays
35 u_prefs, v1_prefs, v2_prefs = init2d(0.,0.), init2d(0.,maint_pref), init2d(0.,maint_pref)
36 e_M, e_G1, e_G2 = u_prefs.copy(), u_prefs.copy(), u_prefs.copy()
37 e_V, V = init1d(0.), init1d(4.68)
38
39 # initialize state
40 h = {'1':0, '2':0, 'A':0, 'B':0, 'C':0, 'X':0, 'Y':0, 'Z':0}
41 s_t = choice(Os) + choice(Os) + choice(Os)
42 s_t_i = Ss.index(s_t)
43 o = s_t[0]
44 wm = [s_t[1], s_t[2]]
45
46 # params
47 #l = eligibility trace decay rate, g = discount rate, a = lrate, r = reward, t = trial
48 l, g, a, r, t = 0.75, 0.94, 0.1, 0.0, 0
49
50 # performance; scr = successive correct responses, n_right = right presses since fail
51 tp, fp, tn, fn = 0.0, 0.0, 0.0, 0.0
52 scr, scrs, max_scr, n_right = 0.0, [], 0.0, 0
53
54 while True:
55     ## Choose motor and gating actions for current state
56     # choose actions for this state
57     u_sm, v1_sm, v2_sm = sm(u_prefs[s_t_i]), sm(v1_prefs[s_t_i]), sm(v2_prefs[s_t_i])
58
59     # actions are [L, R], [U, M]
60     u, v1, v2 = choice([0,1],p=u_sm), choice([0,1],p=v1_sm), choice([0,1],p=v2_sm)
61
62     ## Update motor and gating action eligibility traces, e_M, e_G1, e_G2
63     # e_M = l * g * e_M, s != s_t

```

```

63 e_M, e_G1, e_G2 = e_M*1*g, e_G1*1*g, e_G2*1*g
64
65 # e_M = 1 - Pr(a/s), s = s_t, a = a_t
66 e_M[s_t_i][u], e_G1[s_t_i][v1], e_G2[s_t_i][v2] = 1-u_sm[u], 1-v1_sm[v1], 1-v2_sm[v2]
67
68 # e_M = - Pr(a/s), s = s_t, a != a_t
69 n_u, n_v1, n_v2 = int(not u), int(not v1), int(not v2)
70 e_M[s_t_i][n_u], e_G1[s_t_i][n_v1], e_G2[s_t_i][n_v2] = -u_sm[n_u], -v1_sm[n_v1], -v2_sm[n_v2]
71
72 ## Get reward and next state
73 # get the indexes of each element in the queue
74 ONE, TWO, A, B, C, X, Y, Z = h['1'], h['2'], h['A'], h['B'], h['C'], h['X'], h['Y'], h['Z']
75 if (X == (t-1) and A > B and A > C and ONE > TWO) or (Y == (t-1) and B > A and B > C and TWO >
    ↳ ONE):
76     if u == 1:
77         r, tp, scr, n_right = 1.0, tp+1, scr+1, n_right+1
78     elif u == 0:
79         r, fn = 0.0, fn+1
80         if scr > max_scr:
81             scrs.append(scr)
82             max_scr = scr
83             f.write(str(t) + "\t" + str(max_scr) + "\t" + str(n_right) + "\n")
84             f.flush()
85         scr, n_right = 0, 0
86 elif u == 0:
87     r, tn, scr = 0.25, tn+1, scr+1
88 elif u == 1:
89     r, fp = 0.0, fp+1
90     if scr > max_scr:
91         scrs.append(scr)
92         max_scr = scr
93         f.write(str(t) + "\t" + str(max_scr) + "\t" + str(n_right) + "\n")
94         f.flush()
95     scr, n_right = 0, 0
96
97 # Update internal state based on previous state, gating action and new observation
98 if v1 == 0: wm[0] = o
99 if v2 == 0: wm[1] = o
100
101 # New trial starts here from the environment's perspective
102 # randomly choose the next state and update history
103 o = choice(Os)
104 h[o] = t
105
106 s_nxt = o + wm[0] + wm[1]
107 s_nxt_i = Ss.index(s_nxt)
108
109 # compute state-value prediction error based on critic's state-value approximation
110 d_t = r + g * V[s_nxt_i] - V[s_t_i]
111
112 e_V *= g*1
113 e_V[s_t_i] += 1
114 e_V_tmp = copy(e_V)
115 e_V_tmp *= a*d_t
116 V += e_V_tmp
117
118 e_M_tmp = copy(e_M)
119 e_M_tmp *= a*d_t
120 u_prefs += e_M_tmp
121
122 e_G1_tmp = copy(e_G1)
123 e_G1_tmp *= a*d_t
124 v1_prefs += e_G1_tmp
125
126 e_G2_tmp = copy(e_G2)
127 e_G2_tmp *= a*d_t
128 v2_prefs += e_G2_tmp
129
130 s_t, s_t_i = s_nxt, s_nxt_i
131
132 t += 1
133
134 # print t, max_scr, scr, s_nxt, u
135

```

```
136     if t % 1000 == 0:
137         f.close()
138         exit()
139
140     # if t % 1000 == 0:
141     #     tp, fp, tn, fn = 0.0, 0.0, 0.0, 0.0
142     # if r == 1.0:
143     #     try:
144     #         precision = tp / (tp + fp)
145     #         recall = tp / (tp + fn)
146     #         accuracy = (tp + tn) / (tp + tn + fp + fn)
147
148     #         print t, "precision: " + str(precision) + "\trecall: " + str(recall) + "\tfmeasure: " +
↪ str(accuracy)
149     #     except:
150     #         pass
```