Connecting Minimal Agent Models: From Finite State Controllers to Tiny Recurrent Neural Networks

Nawaf Alkhalifa, Hala Alshammari, Giorgio Nicoletti, Antonio Celani

nalkhali@ictp.it, halshamm@ictp.it, gnicolet@ictp.it, acelani@ictp.it

Abstract

There are two established ways to write down a small, readable model of how an animal or a person makes repeated decisions. A Finite State Controller (FSC) stores what matters as a handful of labelled memory states and a lookup table. A tiny recurrent neural network (RNN) stores the same thing as a short list of numbers updated by trained weights. The two look unrelated, and this project asks how they connect. We fit both to the same synthetic win-stay/lose-shift data, where the correct rule is known in advance, and compare what each one holds inside. The central result is that matching behavior is not the same as learning the rule: on every trial the data contains, the two models are interchangeable, yet a probe that steps just outside the data shows the trained network never learned win-stay at all. It learned a close approximation that slowly drifts to the wrong answer when pushed. We then show, by solving for the network's weights rather than training them, that a network can hold the rule exactly, down to a single hidden unit, and we identify why training misses that solution. The work turns two loose intuitions into precise statements: exactly how much memory a network needs to mirror an M-state controller, and where a softmax readout cannot reproduce a controller's policy.

Date: August 17, 2026

Code: https://github.com/SmartieBrainz/FSC-Experimentations

Contents

1 Introduction 2

2 What we found 2

3 Related Work 3

3.1 Decoding behavior with minimal and interpretable agent models . . . 3

3.2 Discovering cognitive strategies with tiny recurrent neural networks . . . 4

3.3 From these papers to our project . . . 4

4 Methodology 4

4.1 Task and data . . . 4

4.2 The shared pipeline that fits both models . . . 5

4.3 Reading out the two internal states . . . 5

4.4 The comparison analyses . . . 6

4.5 Solving for the network instead of training it . . . 6

5 Results 7

5.1 On ordinary trials the two models are the same . . . 7

5.2 How much memory each model uses . . . 7

5.3 The two models share a representation, one way . . . 7

5.4 The probe that separates them: off-manifold drift . . . 8

5.5 The deterministic solve is exact down to one hidden unit . . . 9

5.6 What the one hidden unit does . . . 10

5.7 The map between the two memories, and the $H \geq M - 1$ condition . . . 10

5.8 Is the solved point somewhere training can stay? . . . 10

5.9 The stochastic controller: the memory transfers, the readout hits a wall . . . 12

5.10 Three states in one hidden unit, without an invertible map . . . 12

6 Discussion 13

7 Limitations and Future Work 14

8 Conclusion 14

A Reproducibility and implementation notes 15

References 15

1 Introduction

An animal on a two-armed task keeps some trace of what just happened and turns it into the next choice. Two modelling traditions write that trace down in different currencies. The first uses a small set of labelled memory states with a table saying which state follows a given action and outcome. This is the Finite State Controller (FSC) of Nicoletti and Celani [1]. The second uses a short vector of real numbers held inside a trained recurrent network, kept as small as the data allows. This is the tinyRNN of Ji-An, Benna and Mattar [2]. Both aim at the same target, a minimal and readable account of how behavior is produced, and both succeed on standard reward-learning tasks. What has stayed open is how one relates to the other.

A short glossary makes the rest of the report readable without prior knowledge of recurrent networks. A *Finite State Controller* is a small machine with a few named memory states; in each state it plays a fixed action, and after it sees the outcome a lookup table tells it which state to move to next. A *GRU* (gated recurrent unit) is the specific small recurrent network the tinyRNN work uses; its memory is a vector of a few real numbers, and trained gates decide, at each step, how much of that memory to keep and how much to overwrite. *Win-stay/lose-shift* (WSLS) is the rule behind our data: repeat a winning choice, switch after a loss. One bit of memory follows it perfectly.

This project studies the FSC-to-GRU relationship on a controlled case where the answer is knowable. Because WSLS needs only one bit, an FSC of two states and a GRU of a few hidden units both have room to hold the rule exactly. Holding the task fixed and simple lets us line the two models up against each other without the confound of a task neither can hold, so any gap that later analysis finds reflects how each one represents the rule, not a shortfall in capacity.

A question drives the work. The FSC is fast, needs little data, and reads directly as a small machine, so it already is a strong model of this behavior. The GRU costs more to fit and hides its computation in continuous weights. Beyond the fact that a network can take continuous inputs, does the extra machinery of the GRU buy anything on a task like this? The stake behind that question is a single point that the rest of the report returns to repeatedly: reproducing behavior and learning the rule that generates it are two different achievements, and a model can do the first without the second.

We use the driving question to organize three phases.

Phase one. builds a shared pipeline that fits both models to the same sessions and reads out their internals on every trial, then validates every stage on WSLS, where the correct model is known.

Phase two. compares the two models. On every measure taken over the trials the data contains they agree; a probe that steps outside the data then pulls them apart, showing the trained GRU does not hold the rule of the data it saw.

Phase three. drops training and solves for the GRU weights directly. This separates a limit of the architecture from a limit of gradient descent and produces the cleanest statements in the report about what a GRU can and cannot be made to do.

The closing sections set out what remains, mainly generalizing the construction beyond two states and carrying minimal agent models into continuous state.

2 What we found

For the reader who wants the result before the machinery, here are the three phases in one paragraph each. The technical versions follow in Sections 3 and 4.

Takeaway

On the win-stay/lose-shift task the two models are the same wherever the data lives, yet the trained GRU never learned the rule. A single-hidden-unit GRU can be made to hold the rule exactly, so the failure belongs to training, not to the architecture. The extra machinery of the GRU buys nothing here and adds a failure mode the FSC does not have.

Phase one, the pipeline. The two papers ship code that expects different data formats and reports internals in different ways. We built one workflow that fits both models to the same sessions and reads out, on every trial, the FSC's belief over its memory states and the GRU's hidden vector. On WSLS the fitted models match the data's stay and switch rates to about 0.002, so the pipeline reproduces both papers' results and gives a fair basis for comparison.

Phase two, the comparison. Every measure taken over the trials the data contains puts the FSC and the GRU within a few thousandths of each other, and the GRU's hidden state even contains the FSC's state as a strict superset. A probe that feeds a model more consecutive wins than any streak in the data then separates them: the FSC answers "stay" with certainty at every length, while the trained GRU's confidence slides toward the wrong answer. It learned a slow transient tuned to the streak lengths it saw, not the rule.

Key Insight

Behavior on ordinary trials cannot tell the two models apart. The separation is visible only off the data, which is why the off-manifold probe is the tool that matters. Any claim that a network "learned the strategy" has to be tested away from the data it was fit on.

Phase three, solving instead of training. We wrote down what a GRU must satisfy to *be* the fitted controller and solved those equations for its weights. A solution exists at every size tested, down to one hidden unit, and it holds the rule exactly. The drift in Phase two is therefore a property of how training arrived at its weights: an L_1 penalty on the recurrent weights slowly erodes the exact solution, and gradient descent from a random start never reaches it. The same construction yields two results that reach past WSLS: a GRU needs at least $M - 1$ hidden units to mirror an M-state controller exactly, and on a stochastic controller a softmax readout cannot reproduce the controller's policy at all beliefs.

3 Related Work

3.1 Decoding behavior with minimal and interpretable agent models

Nicoletti and Celani [1] reconstruct the decision process of a biological agent directly from its behavior, without prior knowledge of its internal workings or its environment. Their agent model holds a small, discrete set of internal states. At each step the agent updates its state from what it observes and picks an action from its current state. The model is fit with a particle-swarm search, MAPSO, combined with gradient descent, and the number of states is chosen by a parsimony argument: the smallest count whose validation loss is within noise of the best.

The authors test the method on synthetic agents first, where the true rule is known, and show it recovers the correct behavior. They then apply it to real data, rats performing an evidence accumulation task and mice making decisions under uncertainty in a changing environment. In

both cases a small number of internal states reproduces the observed choices with high accuracy. Because the model is discrete, each state reads directly, and the authors show what each state means in terms of the animal's recent history, which is the main appeal of the approach.

3.2 Discovering cognitive strategies with tiny recurrent neural networks

Ji-An, Benna and Mattar [2] take a different route to the same kind of question. In place of a discrete state machine, they fit a tiny recurrent neural network, a GRU, to the choices of an individual animal or person. The idea is to keep the network as small as possible, often one to four hidden units, so it stays possible to interpret. Model size is chosen with a nested cross-validation procedure that trains many networks per candidate size and picks the smallest size whose performance is not meaningfully worse than larger ones.

Across six reward-learning tasks the authors show these tiny networks predict choices as well as much larger networks, and better than classical cognitive models of the same size. Because the network is so small, its hidden state can be studied with tools from dynamical systems, which lets the authors read details usually hidden inside a black box.

3.3 From these papers to our project

Both methods recover a small, readable model of behavior: the FSC with discrete states and explicit transitions, the tinyRNN with a few continuous hidden units. The FSC framework assumes discrete actions and observations, while many real tasks involve continuous, high-dimensional variables. Our project starts from the correspondence between an FSC and a linearized GRU noted in [1]: a small RNN can reproduce the FSC's computation and, through function approximation, also handle continuous inputs. The longer aim is to use small RNNs as a compact, interpretable stand-in for the discrete FSC in continuous decision spaces. Before that extension we build and validate the shared pipeline on discrete behavior where the correct model is known, which is what this report describes.

4 Methodology

4.1 Task and data

The behavior is a two-armed choice with a binary outcome on each trial. On every trial the agent picks one of two arms and either wins or loses. The rule behind the data is win-stay/lose-shift: if the last choice won, pick it again; if it lost, switch. One bit of memory covers the rule exactly.

We use five synthetic datasets that span a range of how deterministic the rule is. Four follow "win means stay" on every trial and differ in how a session starts and in how the losing side behaves. The fifth, `rwsls_det_init`, is a fully stochastic control: it stays after a win about 66.5% of the time and switches after a loss about 20.1% of the time, so matching it means landing on those rates rather than on a hard 0 or 1. Each dataset holds one thousand sessions of variable length, close to two hundred thousand trials in total. Table 1 lists them and the role each plays.

Table 1 The five datasets. "Stay after win" is the true rate the data was generated with; a deterministic rule fixes it at 1.00. The longest streak is the longest run of consecutive wins anywhere in that dataset's two hundred thousand trials, and it sets how far the off-manifold probe of Section 5.4 has to reach before it leaves the data.

Dataset	Rule	Stay after win	Longest streak	Role
`original`	deterministic	1.00	28	the tinyRNN paper's WSLS set
`det_init`	deterministic	1.00	33	Phase-three analytical case
`random_init`	deterministic	1.00	28	main comparison dataset
`lwsls_det_init`	deterministic	1.00	28	loss-side variant
`rwsls_det_init`	stochastic	0.67	18	control: no hard certainty

Fixing the data to a WSLS task removes any uncertainty about capacity. An FSC of two states and a GRU of a few hidden units both have room to hold the rule exactly, so any gap that later analysis finds reflects how each one represents the rule.

4.2 The shared pipeline that fits both models

The two papers ship code, but the codebases expect different data formats, define modules that collide by name, and report their internals in different ways. The pipeline stitches them into one workflow. It takes data as a list of sessions, each a small dictionary of `actions`, `observations`, and an optional `reward`, with sessions allowed to have different lengths. This format is our own intake layer; the pipeline converts it into each repository's native format at the point of use, so a user can bring a list of dictionaries, a long-format CSV with one row per trial, or a small synthetic generator and feed the same downstream code.

Both repositories are used as cloned rather than reimplemented, so nothing about how the models are fit is invented here. The FSC is fit with the reference MAPSO search followed by gradient refinement, with the state count chosen by the same parsimony argument the authors use. The GRU is fit with the tinyRNN nested cross-validation procedure, sweeping the hidden size and selecting the smallest size not significantly worse than larger ones. The exact patches applied to the cloned code, the module-switching mechanism, and the caching that makes a fresh checkout reproduce the same run are described in Appendix A; none of them change how either model is fit.

4.3 Reading out the two internal states

Comparing the two models on the same trials needs a readout of what each one holds in memory at each step. The FSC belief is a distribution over the M memory states, computed by the same forward-filtering recursion the library uses inside its loss, so the readout is the model's own inference rather than a new one. The GRU hidden state is a vector of H numbers, taken from one forward pass.

To see inside a GRU step we also reimplement one step in plain numpy, which exposes the three gates the later analysis needs and is checked against the library step for step. One GRU step reads

$$r = \sigma(W_{ir}\, x + b_{ir} + W_{hr}\, h + b_{hr}), \tag{1}$$

$$z = \sigma(W_{iz}\, x + b_{iz} + W_{hz}\, h + b_{hz}), \tag{2}$$

$$n = \tanh(W_{in}\, x + b_{in} + r \odot (W_{hn}\, h + b_{hn})), \tag{3}$$

$$h' = (1 - z) \odot n + z \odot h, \qquad P(\text{action}) = \text{softmax}(W_{\text{lin}}\, h' + b_{\text{lin}}), \tag{4}$$

where x is a four-number one-hot input that encodes the last arm and the last outcome (a one-hot uses one slot per category, set to 1 for the active category and 0 elsewhere), $\odot$ is the elementwise product, and σ is the logistic function that squashes any number into $(0, 1)$. Read the three gates in plain terms. The *reset gate* r decides how much of the current memory feeds into the proposed new memory n. The candidate n is that proposed new memory. The *update gate* z is the blend knob between old and new: at $z = 1$ the state is held unchanged, $h' = h$; at $z = 0$ it is overwritten, $h' = n$. That reading of z carries most of the later results. The final line is the *readout*: it turns the hidden state into a probability over the two actions through a softmax, the standard map from a vector of scores to a probability distribution.

One asymmetry runs through the whole comparison. The FSC belief is a probability distribution over named discrete states; the GRU hidden state is an unlabelled point in continuous space. The two are not the same kind of object, and turning that from a problem into a result is what the analyses below are for.

4.4 The comparison analyses

Four instruments carry the comparison, each applied to both models on the same trials.

Behavioral rates and the action matrix. The stay-after-win and switch-after-loss rates come from replaying real sessions and averaging the probability a model assigns to the repeat move and to the switch. The action-transition matrix $P(A_t = a' \mid A_{t-1} = a,\ Y_{t-1} = y)$ is the finer version of the same measurement: instead of collapsing to two numbers, it reports each model's predicted action probability averaged over all trials that share the same last action-and-outcome pair. Both summaries are built the same way for the two models, so any gap between them reflects the models and not the measurement.

Effective dimensionality. The participation ratio of the hidden-state PCA spectrum counts how many independent directions of the hidden space the GRU actually moves along, regardless of how many units it has. PCA (principal component analysis) finds those directions in order of how much the state varies along each. A leave-one-out freezing test, pinning one unit to its mean and replaying sessions, checks whether a given hidden unit contributes to the final output at all.

Linear probe. A logistic-regression classifier predicts one model's state from the other model's hidden readout, scored by its own five-fold cross-validation. Because a linear probe draws only straight boundaries, a high score means the information sits inside the representation rather than near it. We run it both ways, GRU state to FSC state and FSC belief to GRU cluster.

Off-manifold probe. The data contains only the situations it happens to contain; the longest run of wins in a row bounds what the models ever saw. Feeding a fitted model a run of identical winning trials longer than any in the data asks whether it holds the rule where the data cannot score it. WSLS fixes the answer at certainty for every run length; the probe reads what the model does instead. This is the instrument that separates the two models, and Section 5.4 is built on it.

4.5 Solving for the network instead of training it

Training a GRU and then measuring the gap to the FSC cannot separate a limit of the architecture from a limit of gradient descent, because failing to find a set of weights is not evidence that none exists. To remove that ambiguity we skip training and solve. The fitted FSC already is a finite-state machine: a set of memory states, a transition table $g(m' \mid y, a, m)$, a per-state action rule $\pi(a \mid m)$, and a start state m_0. We ask for the GRU weights that make the network be that machine exactly.

Each memory state m is placed at some hidden vector $\vec{e}_m$, and the placements are collected as columns of a matrix E. The embedding is not known in advance; the solver finds it alongside the weights, because a placement of the states and a set of weights only work together. Three families of requirement pin the network, with $x(a, y)$ the one-hot input and $\mathrm{GRU}(\vec{h}, \vec{x})$ one step:

$$\underbrace{\mathrm{GRU}(\vec{e}_m,\, x(a,y)) = \vec{e}_{g(y,a,m)}}_{\text{transitions, every reachable } (m,a,y)} \qquad \underbrace{\mathrm{softmax}(W_{\mathrm{lin}}\vec{e}_m + b_{\mathrm{lin}}) = \pi(\cdot \mid m)}_{\text{readout, every } m} \qquad \underbrace{\vec{e}_{m_0} = \vec{0}}_{\text{start}}. \tag{5}$$

Land where the table says, play what the state says, and start where tinyRNN starts, at $h_0 = \vec{0}$. The first block carries the weight: it asks the table to hold as an identity in hidden space, on every reachable situation and not only on average, which is exactly what the trained network fails.

Each requirement becomes a residual, the quantity "what the network does" minus "what it must do", which reads zero when the requirement holds. A least-squares solver drives every residual to zero at once. Because the readout target is a hard one-hot that a softmax reaches only with infinite weights, the target is pulled a small amount off the vertex,

$$\tilde{\pi}(\cdot \mid m) = (1 - \mathtt{conf})\,\pi(\cdot \mid m) + \frac{\mathtt{conf}}{A}, \qquad \mathtt{conf} = 10^{-3}, \tag{6}$$

which is the only approximation in the construction. A softmax never reaches a hard 0 or 1, so the target sits a hair off the corner; this sets the 0.9995 ceiling that appears later. There are more unknowns than requirements, so many networks satisfy the list; the solver runs from several random starts and keeps the smallest-weight answer, which makes the result reproducible rather than arbitrary. Nothing is trained.

5 Results

5.1 On ordinary trials the two models are the same

Every model matches the data's stay and switch rates to about 0.002 or better, and the FSC often lands on exactly 0 because it uses a hard rule. The action-transition matrix agrees to three decimals between the FSC and the GRU. For the FSC the matrix is essentially exact by construction, since its transition already is a function of the last action and outcome; for the GRU it is a genuine summary that comes out clean to the extent the network has collapsed to a one-step rule, and on WSLS it does. Nothing on the trials the data contains separates the two models, which is the point: the separation has to be looked for elsewhere.

5.2 How much memory each model uses

The FSC selects $M^* = 2$ states on every dataset, both states heavily used and split near 50/50, each lining up cleanly with one of the last action-and-outcome contexts. The GRU's selected hidden size swings across datasets, from two units up to eight, five on the main `random_init` dataset and eight on `det_init`. That swing is a property of the size-selector, not of the task: the participation ratio stays in a tight band, about 1.7 to 2.9 independent directions, on every dataset and at every selected size. A GRU handed eight units runs on roughly as much memory as one handed two. The selected hidden size is not a reliable measure of task complexity; the participation ratio is, and it sits close to the FSC's $M^* = 2$ every time. Extra units buy redundancy, not new content.

5.3 The two models share a representation, one way

The linear probe is sharply asymmetric. From the GRU hidden state the FSC state is recovered on essentially every trial, near 100% against a 50% chance line. From the FSC belief the GRU cluster

is recovered only up to the correct pair of clusters, capped at the data's win rate, because the FSC merges each winning context with the losing context that implies the same next move. The GRU hidden state is a strict superset of the FSC state: it holds everything the FSC keeps, plus the extra win-and-lose detail the FSC throws away. On WSLS that extra detail buys nothing, since both models predict behavior equally well and the FSC's two states are enough, cheaper, and easier to read.

5.4 The probe that separates them: off-manifold drift

The longest run of wins in a row anywhere in a dataset is about 28, so a run of 50 wins never appears, and whatever a model does out there costs training nothing. Started from a blank memory and fed "you picked arm A and you won" for k steps, WSLS fixes the correct answer at a flat line at 1.0 for every k. The FSC draws that line, its transition being a table lookup with no notion of accumulated evidence. The trained GRU does not (Figure 1). It holds near 1.0 across the streak lengths the data carries; at $k = 11$, where 99% of all win streaks have ended, it still reads 0.95 or better, and then it slides. By $k = 28$, the single longest streak in two hundred thousand trials, the failing arm is already down to about 0.71, and pushing further it falls through 0.5 and out the other side, reaching about 0.23 at $k = 80$.

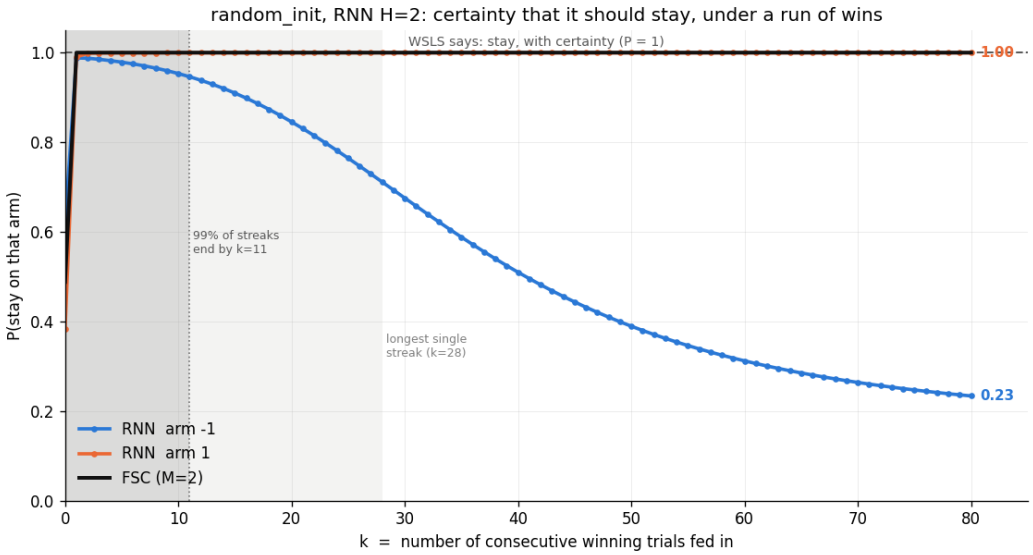


Figure 1 The off-manifold probe. Fed a run of k consecutive wins on one arm, the FSC (black) reports certainty that it will stay, at every length. One arm of the trained GRU (blue) holds near 1.0 across the streak lengths the data contains (shaded), then slides toward the wrong answer once the input runs past anything the data held. This is the one measurement that tells the two models apart.

The mechanism reads directly off the GRU update. Feeding the same input forever turns the GRU into one fixed map applied over and over, and setting $h' = h$ collapses the update to a resting state that is whatever the input proposes; the update gate z acts as a brake on how fast the state arrives, not on where it arrives. So the resting readout is $\mathrm{softmax}(W_{\mathrm{lin}} n^* + b_{\mathrm{lin}})$, and nothing in the architecture pins it at 1.0. On the failing arms that resting readout sits near 0.1 to 0.2, where WSLS demands 1.0, and the pull toward it is weak, spectral radius (the factor by which the map contracts each step) about 0.96 to 0.99. Slow is what lets training miss it: at $k = 11$ the readout still says 0.95, so there is almost nothing for the loss to charge, and the one longest streak at $k = 28$ is a single trial out of two hundred thousand.

Two controls close off the easy explanations. Capacity is not the story: at two and four hidden units exactly one arm drifts, a different arm each time, decided by the random weights the network started from, and only at eight units do both arms hold on the baseline fits. Regularization does not fix it either; it makes it worse. A heavier L_1 penalty on the recurrent weights, 10^{-3} against the 10^{-5} baseline, drifts further in five of six dataset-and-size cells, sending the eight-unit `random_init` fit from 0.98 down to 0.52. The one clean pass is the stochastic control `rwsls_det_init`, whose true rule stays after a win about 67% of the time; there the GRU settles on 0.67, because the correct answer is a soft rate it can rest on rather than a hard certainty it cannot hold. Figure 2 shows the pattern across all five datasets.

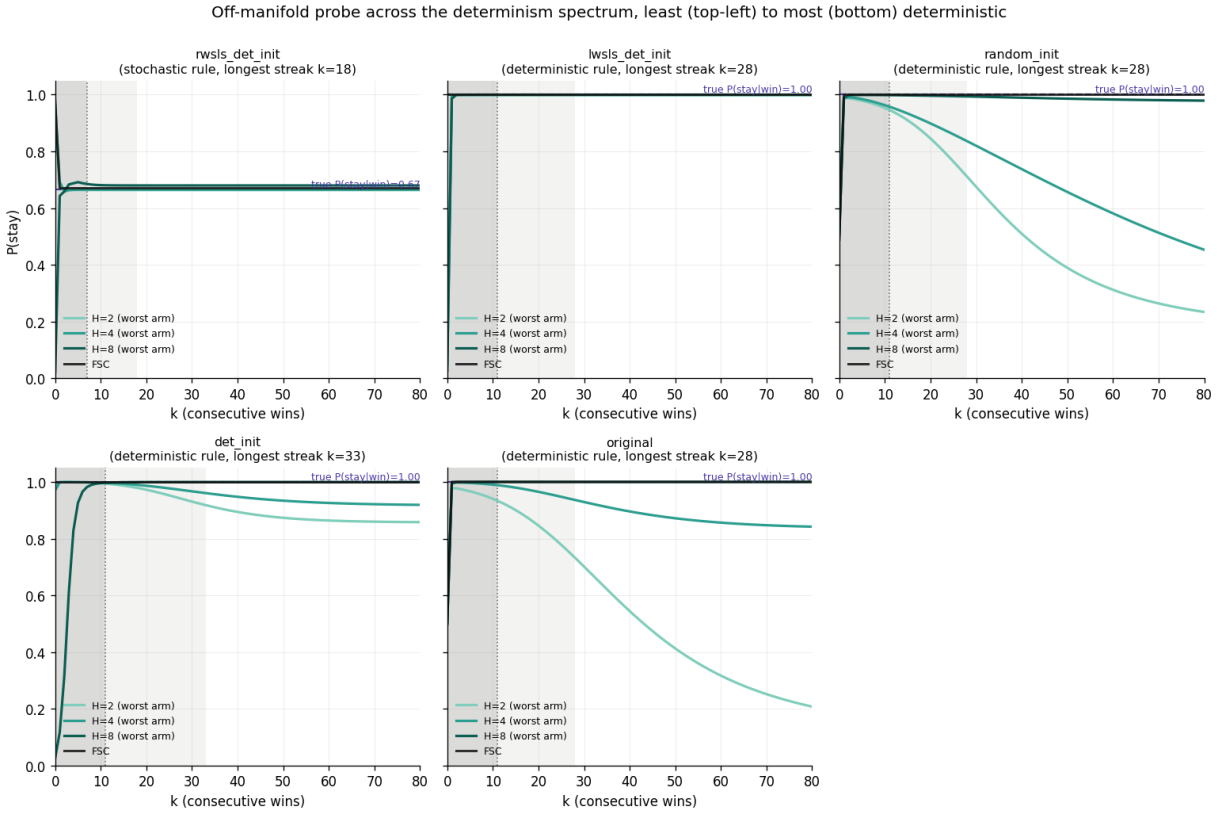


Figure 2 The probe across the five datasets, ordered from least deterministic (top left) to most (bottom). The drift appears wherever the true rule is a hard certainty and the input runs past the data, and it disappears on the stochastic control, whose correct answer is a soft rate the network can settle on. It is not tied to one dataset, and it is not universal.

Matching behavior and learning the rule are two different achievements. The FSC represents the rule as a lookup, so "you won, so stay" holds no matter how much evidence piles up. The GRU learned an approximation that is right where the data lives and drifts to the wrong certainty outside it. What in training produces the drift, and whether it could be trained away, would need a rigorous hyperparameter study, out of scope here in both time and computing budget. The third phase takes a sharper route instead.

5.5 The deterministic solve is exact down to one hidden unit

On the deterministic `det_init` machine the solver of Section 4.5 reaches every requirement of Equation 5 to within 8×10^{-9} at a single hidden unit and about 6×10^{-10} at two and three units. Replayed through the untouched numpy GRU over one thousand real sessions, the solved

network sits on the FSC's memory state with a mean error near 10^{-9} per trial and picks the same action on 100% of trials. The network is not an approximation of the controller; it is the controller, written in continuous coordinates instead of discrete ones. So the drift of Section 5.4 is not a capacity limit of a GRU this small. That GRU can hold the rule exactly. The drift is about how training arrived at its weights.

5.6 What the one hidden unit does

At a single hidden unit the machine is small enough to read completely. The FSC has two states, each plays one arm with probability one, and there are two outcomes, so the network is ever in only $2 \times 1 \times 2 = 4$ situations on the data, and those four rows are the whole behavior. Counting over the real sessions, the solved one-unit network takes exactly two hidden values, one per memory state. The mechanism is a single gate. In the solved weights a win adds about +11.3 to the update gate's input and a loss subtracts about 8.7, which dwarfs everything else in the row, so on a win $z \to 1.0000$ and the state is held, and on a loss $z \to 0.1$ and the state is overwritten. Win-stay/lose-shift comes out as hold on a win, reset on a loss. Nothing told the solver to use the update gate this way; it fell out of requiring the network to be the machine.

5.7 The map between the two memories, and the $H \geq M - 1$ condition

The two models store memory differently: the FSC holds a distribution $\vec{p}_{\text{FSC}} \in [0,1]^M$ over discrete states, the GRU a vector $\vec{h} \in \mathbb{R}^H$. We ask whether one is an affine translation of the other in both directions, $\vec{p}_{\text{FSC}} = A\vec{h} + \vec{b}$ and $\vec{h} = B\vec{p}_{\text{FSC}} + \vec{c}$; an affine map is a linear map plus a constant shift. One direction comes for free: a belief is a mixture of states, so the hidden vector that represents it is the same mixture of the columns of E, giving $\vec{h} = E\,\vec{p}_{\text{FSC}}$, that is $B = E$ and $\vec{c} = \vec{0}$. The other direction, recovering the belief from the hidden state, needs $A\vec{e}_m + \vec{b} = \vec{u}_m$ for every state, where $\vec{u}_m$ is the one-hot of state m. Stacking that over the M states gives

$$\begin{bmatrix} \vec{e}_0^{\top} & 1 \\ \vdots & \vdots \\ \vec{e}_{M-1}^{\top} & 1 \end{bmatrix} \begin{bmatrix} A^{\top} \\ \vec{b}^{\top} \end{bmatrix} = I_M, \tag{7}$$

which has an exact solution when the matrix on the left has rank M, that is, when the embedded states sit at genuinely different places, none lying on the flat spanned by the others. Reaching rank M takes enough hidden dimensions to spread the states apart, which needs

$$H \geq M - 1. \tag{8}$$

The bound is one below the obvious guess of $H = M$, because a distribution over M states has only $M - 1$ free numbers once its entries are required to sum to one. The offset $\vec{b}$ is not optional: the start requirement puts one embedded state at the origin, so a purely linear map would send $\vec{h} = \vec{0}$ to $\vec{p} = \vec{0}$, which is not a distribution. The solve instead finds $\vec{b} = \vec{u}_{m_0}$, the one-hot of the start state. With $M = 2$ the map is exact at $H = 1$, reconstruction error at machine precision.

5.8 Is the solved point somewhere training can stay?

Starting from the solved weights and training under cross-entropy alone, the answer is yes: two thousand epochs leave the off-manifold drift at 10^{-8}, where it started, while the loss falls by orders of magnitude. Under the objective that produced the committed fits it flips. That objective adds an L_1 penalty of 10^{-5} on the recurrent weights, and the penalty slowly eats the solution: the update gate's "hold" slips from 0.99999997 toward 0.995, and by two thousand epochs the rule

is broken off the manifold while on-manifold accuracy stays pinned at 100% and the loss keeps falling the whole way down (Figure 3). Switching the penalty off and changing nothing else stops the erosion, which pins the penalty as the cause. The conflict is structural: saturating a gate takes a large weight, and L_1 is a charge on exactly that, so the pipeline's own regularizer prices the exact solution out, and it does so silently.

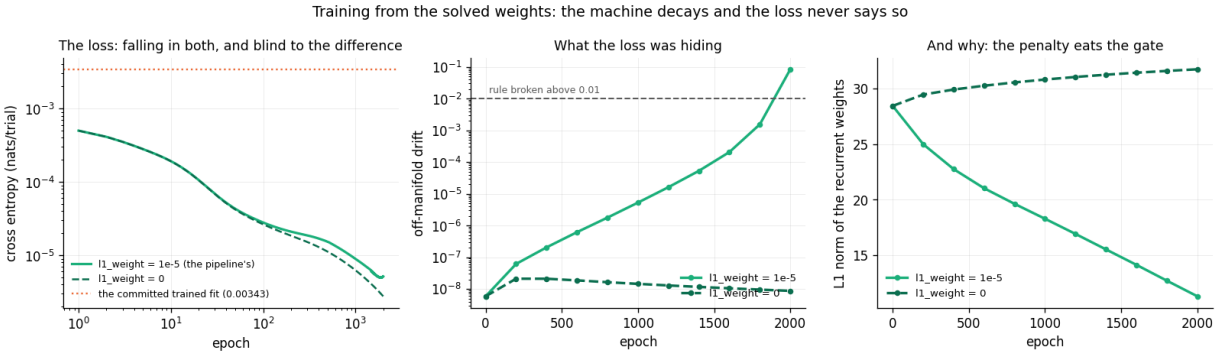


Figure 3 Training the one-unit GRU from the exact solved weights on `det_init`. The loss falls in both runs and cannot see the difference (left). With the pipeline's L_1 penalty on, off-manifold drift climbs past the broken-rule line while accuracy stays at 100% (middle), because the penalty shrinks the large recurrent weight that holds the gate saturated (right). With the penalty off, the solution stays put.

That raised the obvious next question. If the L_1 penalty is what eats the solution, does a cold train from random weights with the penalty set to zero recover the rule? On `det_init` it does not. Three random starts at zero penalty reach 100% accuracy on the data and still get the rule wrong off it, with final drift 0.13, 0.12, and 0.62, the same as the committed fits. So the drift is not only the penalty; it is also where the optimizer starts. The exact solution is reachable and stable once you are on it under cross-entropy, but gradient descent from a random start does not walk to it. A perturbation ladder bounds how far the solved region reaches: knocking the weights off by $\theta_0 = \theta^* + \varepsilon\,\xi$ and retraining, perturbations up to $\varepsilon = 3$, which move the weights by more than half their own length, are undone by every seed, while the same optimizer from a random init at the same budget never gets there (Figure 4). The committed fits' drift is not a sign that the solved region repels gradient descent, and it is not an epoch-budget problem; it is about where the optimizer starts, plus a regularizer that charges rent on staying.

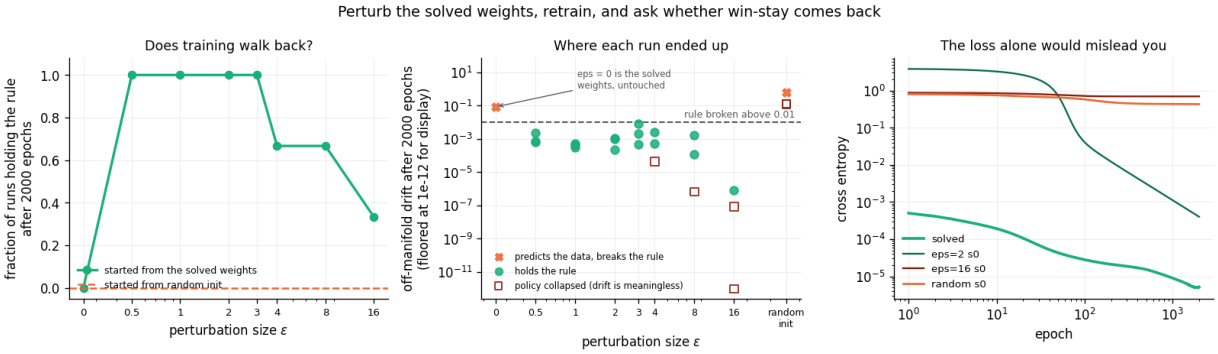


Figure 4 Damage the solved weights by a random step of size ε and retrain. Training walks back to the rule from knocks as large as the whole weight vector (left, green), while a random init at the same budget never holds the rule (left, dashed). The loss alone would not reveal this (right): a run that predicts the data perfectly can still hold the wrong rule off the data.

5.9 The stochastic controller: the memory transfers, the readout hits a wall

The deterministic solve quietly relied on the belief sitting exactly on a memory state on every trial, so pinning the M states pinned the whole trajectory. On the stochastic control the belief is usually a mixture like 70/30 and sits fully on one state on only 18% of trials, so the M corners are a measure-zero part of the path. Run unchanged, the original construction still solves every requirement to machine precision and is no longer the controller: over real sessions its state sits about 10^{-2} from the FSC's, against 10^{-9} on the deterministic task, because the requirements only pinned the network where the belief almost never is.

The memory is repaired by stating the requirement along the whole belief line instead of at its two ends, sampling about eleven belief points across the simplex and targeting the Bayesian filtering posterior rather than the forward transition. That drops the tracking error by roughly 3500 times, to 10^{-6}, at every hidden size including one unit. The readout does not repair the same way. The FSC's action probability is linear in the belief, a straight line as the belief slides from one state to the other, while a softmax over a linear embedding of the belief is a curved line, and a line and a curve cross at a couple of points and cannot coincide (Figure 5). The gap reaches about 0.08 across the beliefs that occur and is identical at one and four hidden units, so it is a shape mismatch, not a capacity limit.

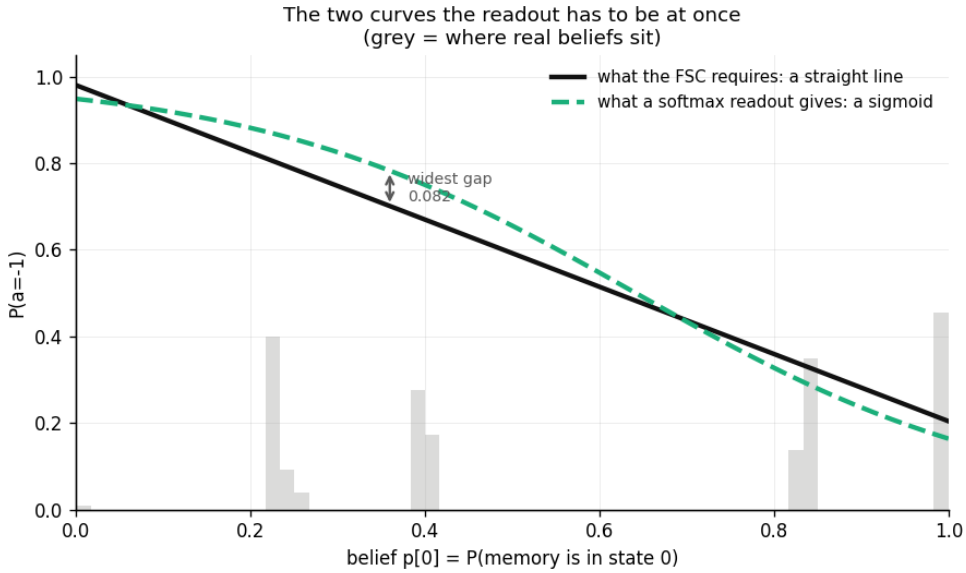


Figure 5 The readout wall on the stochastic controller. The FSC needs the action probability to be a straight line in the belief (black); a softmax over a linear embedding can only draw a curve (green), and the two cross at two points rather than coincide. The widest gap over the beliefs that actually occur (grey) is about 0.08, the same at one and four hidden units.

One consequence is worth stating plainly: on this stochastic task a trained network slightly beats the exact construction on held-out cross-entropy per trial, 0.5655 for the best trained network against 0.5762 for the repaired construction and 0.6195 for the original one, because training is not bound to place the states linearly and so its readout is not forced into the wrong shape. That inverts the deterministic result, where the construction was exact and training drifted.

5.10 Three states in one hidden unit, without an invertible map

The $H \geq M - 1$ condition of Section 5.7 only bites once M grows past two, so we ran the solver on a three-state machine at hidden sizes one, two, and three. At a single hidden unit the network still

solves every behavioral requirement, worst violation at 10^{-9}, and it predicts correctly, because a softmax can turn three points sitting on one line into three different action distributions. So a one-unit GRU reproduces a three-state machine's behavior. What it cannot do at one unit is hold the memory as a recoverable state. The three embedded states land within about 10^{-3} of each other on a single line, the stacked matrix of Equation 7 has rank 2, one short of the $M = 3$ it needs, and no affine map recovers the state, reconstruction error about 0.67 (Figure 6). At two hidden units the three states spread into a plane, the rank reaches 3, and the map inverts to machine precision.

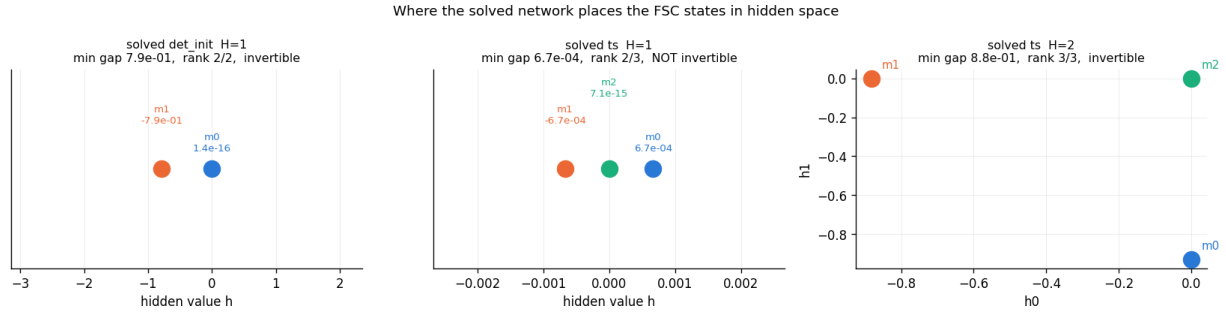


Figure 6 Where the solved network places the memory states in hidden space. A two-state machine fits on a line at $H = 1$ and the map inverts (left). A three-state machine at $H = 1$ collapses its three states onto nearly one point, rank 2 against the $M = 3$ needed, so no affine map recovers the state (middle). At $H = 2$ the three states spread into a plane and the map inverts (right). This is the $H \geq M - 1$ condition in a picture.

This is the concrete case that separates two claims that sound the same: reproducing a machine's behavior and being in exact correspondence with its memory. Behavior needs only enough softmax nonlinearity; the invertible correspondence needs $H \geq M - 1$, and for three states one hidden unit is one short.

6 Discussion

The comparison sits on a task small enough that the correct answer is known in advance. With one bit of memory as the target, an FSC of two states and a small GRU both have room to hold the rule exactly, so the interesting question is never capacity; it is representation. On every behavioral measure that lives on the data, predictive likelihood, the action-transition matrix, the stay and shift rates, the FSC and the GRU are indistinguishable, and the GRU's hidden state even contains the FSC's state as a strict superset. Read on ordinary trials, the GRU's extra machinery buys nothing on WSLS.

The separation shows up only off the data. The trained GRU implements win-stay as a slow transient tuned to the streak lengths it saw, not as a fixed rule, so under a run of wins longer than anything in the data its confidence slides toward the wrong certainty. The FSC has no way to fail like this, because its winning state points at itself and its resting answer is the policy value after one lookup.

The two design choices that carry the third phase are what let us say this cleanly. Solving for the network instead of training it removes the ambiguity between a limit of the architecture and a limit of gradient descent, and the answer is unambiguous: a GRU down to a single hidden unit can be the controller exactly, so the drift belongs to training, specifically to a random starting point and an L_1 penalty that charges rent on the large weights a saturated gate needs. The off-manifold probe is what turns "the two models match" into a testable claim, because it is the one place a

model that copied the data can be told apart from one that holds the rule.

The answer to the driving question, on this task, is that the GRU's extra complexity does not translate into anything useful and in fact adds a failure mode the FSC does not have. That is a statement about WSLS, a one-bit rule, and it is exactly the setting where the FSC should win. The construction also produced two results that reach past WSLS: the $H \geq M - 1$ condition, which says when a GRU and an M-state controller can hold an exact two-way correspondence, and the readout wall on the stochastic controller, which says a softmax over a linear embedding cannot report a linear-in-belief policy exactly. Both are statements about shape that do not depend on the rule being one bit.

7 Limitations and Future Work

The worked case is one rule family on synthetic data. WSLS needs a single bit, so the two-state FSC and the small GRU are both comfortable, and the analysis here does not carry to tasks that demand more memory. Every result off the data is measured on a probe of pure winning inputs; a perturbation of the same size chosen against the network could break it earlier, so the recovery boundary in Section 5.8 is a robustness statement, not a basin radius. The three random cold-train runs are a control, not a search, so they show that the fits which drifted did not reach the solved region, not that gradient descent cannot reach it in principle.

The analytical construction is exact but not robust. The held states are neutral, not attracting, so nothing restores the hidden state if it is knocked off, and it is precisely the trained network's contraction, the property that buys robustness, that also causes the drift. The repair for the stochastic controller is written for two states, a segment; a larger M needs a grid over the simplex, which is a change of sampling rather than of method and is not verified here. The readout wall is argued for the softmax-over-affine readout that tinyRNN uses; whether another common readout could match a linear-in-belief policy is open.

Several threads follow directly. The three-state solver in Section 4.10 is the first step past two states, and running it on a machine whose states cannot be read off the last trial is where the $H \geq M - 1$ condition becomes a real constraint and where the question of whether training finds the solved solution becomes worth asking with something at stake. The question the second phase raised and set down, why gradient descent misses a solution that sits in the space it searched, is open and would need the hyperparameter study that was out of scope here.

The longer aim carries minimal agent models from discrete states into continuous ones. The FSC is discrete by construction; a small recurrent network is continuous by construction and can be solved to match a discrete controller, which makes the network a candidate route to synthetic agents whose state lives in a continuous space, where a finite table of states no longer fits. Function approximation is the technical phase that follows.

8 Conclusion

On a task where the correct model is known, a small discrete controller and a small recurrent network produce identical behavior while computing different things underneath. The FSC represents the rule as a lookup; the trained GRU approximates it wherever the data lives and drifts to the wrong certainty outside it. Solving for the weights rather than training them shows the drift is not a limit of the architecture, since a single hidden unit can be the controller exactly, but a limit of how training arrives at its weights. Two results outlast the one-bit task: a GRU needs at least $M - 1$ hidden units to mirror an M-state controller in exact two-way correspondence, and a softmax readout cannot reproduce a stochastic controller's linear-in-belief policy at every belief.

The practical lesson is narrow and firm: a network that matches behavior has not thereby learned the rule, and only a test off the data can tell the two apart.

A Reproducibility and implementation notes

This appendix records the engineering that keeps the pipeline reproducible. None of it changes how either model is fit; it is separated from the methodology so the scientific narrative there stays uninterrupted.

Vendored repositories. The two upstream codebases are not committed to this project. A setup cell clones each one fresh at a shallow depth into local folders, so a checkout of this project is the notebook, the data file, and the cached fitted models. Running the setup cell once makes the libraries appear locally.

Two automated patches. The setup cell applies exactly two changes to the cloned tinyRNN code, both idempotent so a fresh clone reproduces them. The first replaces an upstream hardcoded absolute save path with a local relative one, so the notebook is portable. The second is a one-line bug fix in the dataset loader: the upstream code indexed a padding mask with a boolean array sized to one session against an axis sized to the longest session, which raised an error on any dataset with variable session lengths. The fix indexes the correct sub-range. Because both patches live in the notebook rather than as hand edits, a fresh clone and run reproduces them exactly.

Module-name collisions. Both repositories define top-level modules with the same names. The code swaps the import path and clears stale modules before importing from either side, so the FSC and tinyRNN imports do not shadow each other within one process.

The plain-numpy GRU step. PyTorch's GRU returns only the hidden state, so one step is reimplemented in numpy to expose the reset, update, and candidate gates the dynamical analysis needs. It is checked against the library step and agrees to machine precision, worst difference about 2×10^{-16} over a full session, so every gate-level result runs against code the solver never touched.

Caching. The FSC state sweep and the GRU nested cross-validation are slow, so both are cached under a per-dataset folder keyed by the run configuration. A fresh checkout reproduces the same run without repeating the slow fits; deleting the cache or flipping the force toggles regenerates it from scratch.

A note on convergence. The committed GRU fits were stopped by the two thousand epoch cap with validation loss still falling and the best epoch being the last one in every run, so those networks had not converged. This matters for reading the Phase-two drift: the trained networks are the ones the tinyRNN procedure produces at its default budget, not fully converged optima.

References

[1] Giorgio Nicoletti and Antonio Celani. "Decoding behavior with minimal and interpretable agent models". In: *bioRxiv* (2026). Preprint. DOI: 10.64898/2026.01.20.700580. URL: https://www.biorxiv.org/content/10.64898/2026.01.20.700580v1.

[2] Li Ji-An, Marcus K. Benna, and Marcelo G. Mattar. "Discovering cognitive strategies with tiny recurrent neural networks". In: *Nature* 644 (2025), p. 993.