Felix Baastad Berg

Spatial Representation Learned in the Recurrent Memory of Artificial Neural Agents in Open-Ended Reinforcement Learning

Exploring Biologically Inspired Memory-Driven Navigation and Implicit Planning.

Masters thesis in Physics and Mathematics
Supervisor: Benjamin Dunn
June 2025

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

NTNU
Kunnskap for en bedre verden

Abstract

Encoding and exploiting spatial knowledge is critical for intelligent foraging, whether in animals or machines. Building on this insight, we created a deep-reinforcement-learning agent whose Long Short-Term Memory (LSTM) core must remember and plan in a two-dimensional grid world where survival is the sole objective. With Proximal Policy Optimization, we trained several variants that differ only in three biologically inspired priors: an auxiliary path-integration head that compels the agent to estimate its own position, a sparsity prior that removes 90 % of weights to mimic economical brain wiring, and a continuous attractor network that supplies an explicit grid-cell basis for metric space.

Our simulations show that pruning the network does not harm reward and that adding the path-integration head markedly boosts performance, letting agents range farther from the origin and return more often to places where positional uncertainty is highest. Introducing a continuous attractor network leaves learning curves unchanged — even though it shares the LSTM's memory budget — yet it still develops hexagonal firing fields and shows signs of encoding other task-relevant variables, such as predator distance. Generalized Linear Models and neural-decoding analyses further reveal that, in sparse path-integrating agents, only a small subset of neurons carries the bulk of spatial information, indicating an efficient representation. The agents also showed an understanding of depletion: the chance of revisiting a patch was lower after eating there, as the area had been depleted of food.

Taken together, these results demonstrate that an artificial agent can navigate and remember with orders of magnitude fewer neurons than many insects while maintaining interpretable, brain-like codes. By deriving and implementing grid-cell dynamics in JAX, we connect models of artificial and biological spatial cognition, enabling systematic comparisons between neural data and supporting future work on attractor-based representations.

Contents

1 Introduction

Animals are hypothesized to create spatio-temporal maps of their environment during foraging, enabling them to remember locations of food sources and predators [1]. This cognitive ability is crucial for survival, as it helps them determine which areas to explore and which to avoid. However, deciphering the long-term neural dynamics and memory underlying such behavior has been proven difficult [subsection 1.1]. One approach to better understand these processes is to develop computational models that mimic the function of the neural circuits found in biological organisms.

To study this setting further, we consider artificial agents navigating a 2D grid-world environment where they must satisfy fundamental needs such as thirst, hunger, and sleep while avoiding predators. Our agent's neural architecture is biologically inspired, incorporating realistic sensory inputs, reward structures and behavioral constrains. Unlike conventional reinforcement learning (RL) agents that rely on explicit positional data, our model leverages memory, aiming for the agent to implicitly remember survival-relevant locations and their relative positions.

This thesis has two main objectives. First, we aim to simulate animal foraging behavior by incorporating biologically plausible priors into reinforcement learning models, and understanding which priors are important to recreate natural foraging behavior. Second, we analyze how spatial information is encoded in the agent's memory. By examining different neural architectures, we seek to qualitatively assess their ability to represent physical space and compare their memory mechanisms — work that potentially can be compared with brain recordings to bridge the gap between artificial and natural intelligence.

1.1 Related work

There already exists a significant amount of research within RL for 2D grid worlds [2, 3, 4, 5, 6]. Works like Tizhoosh [7] and SunWoo and Lee [6] have explored path search and navigation in grid worlds using RL, focusing on performance optimization and environment-specific tuning.

The most commonly cited paper for LSTM was published in 1997 [8] and LSTM has since been an area of active research [9, 10, 11]. The introduction of LSTM revolutionized sequence modeling by enabling networks to capture temporal dependencies. This concept has been widely adopted in RL, particularly in tasks requiring memory or temporal context. More recent works, such as those by Grzelczak and Duch [4], integrate deep RL with LSTM architectures for path planning and memory modeling. However, most existing work on biology-inspired navigation involves small, fully observable arenas with few-to-no obstacles, partly because recording neural activity in naturalistic settings is challenging.

Grid cells were first reported in the medial entorhinal cortex by Hafting *et al.* (2005) [12]. A large body of theory treats this code as a two-dimensional *continuous attractor*: a recurrent network whose activity bump translates in proportion to integrated velocity. The seminal model of Burak & Fiete (2009) showed that such networks can maintain accurate path integration over behaviourally relevant timescales [13], and later work has added realistic spiking dynamics and robustness to noise (e.g. Ságodi *et al.*, 2024) [14].

In parallel, deep-learning studies have demonstrated that grid-like representations can *emerge* or be *programmed* in task-optimised networks. Cueva & Wei (2018) found that training recurrent networks for self-localisation from velocity cues alone yields units with hexagonal firing fields [15]. Banino *et al.* (2018) extended this idea to actor–critic agents, showing that imposing a grid-like latent space improves 3-D navigation performance [16].

A particularly relevant work is Craftax [17], a recent benchmark designed for open-ended reinforcement learning. Craftax builds upon Crafter [18], but significantly expands its complexity and computational efficiency. By implementing a JAX-based architecture (see subsubsection 4.6.2 for introduction to JAX), Craftax enables large-scale reinforcement learning experiments to run orders of magnitude faster than prior benchmarks, making it feasible to study long-term planning, exploration, and memory-driven behaviors without excessive computational costs. Notably, Craftax introduces multi-floor environments, a diverse set of enemies, and mechanics such as enchantments, potions, and a hierarchical progression structure. These features make it an appealing environment for evaluating agents that must integrate past experiences, manage survival constraints, and develop strategies dynamically. Given its emphasis on memory and long-term reasoning, Craftax closely aligns with our goal of developing biologically inspired agents that navigate and

adapt in complex environments. Our work heavily modifies Craftax to be suitable for neuroscience research, including generating natural environments.

1.2 Contributions

During my research stay at Harvard University (Fall 2024), I collaborated with the Rajan Lab on initial statistical analyses of spatial representations, testing neural-decoding scripts and single-neuron metrics. That groundwork shaped my project thesis [19]. For the master's thesis I extend neural architectures built on code originally developed by the Rajan Lab, and I have done extensive additional decoding. Thus, the work presented here is a joint effort: my contributions build on their foundations, guided by the team's continual advice and support.

This work advances our understanding of memory-driven reinforcement learning in biologically inspired agents by demonstrating how RL can enable efficient spatial navigation and decision-making in complex environments.

We show that agents trained via RL successfully explore arenas to locate resource-rich patches and strategically travel between previously visited locations, even when these locations are outside the agent's current field of view and have remained unobserved for hundreds of timesteps. Remarkably, this level of navigational planning is achieved with neural architectures that are significantly smaller than those of insects (including fruit flies [20, 21]), with sparse connectivity constraints similar to those found in insect brains.

To better understand the role of memory in navigation, we analyze the hidden states of the agent's neural network, performing both neural and behavioral decoding to investigate how spatial information is stored and utilized. This allows us to evaluate whether simple memory mechanisms and reinforcement learning alone are sufficient for effective navigation in large, partially observable foraging environments.

Additionally, we examine the impact of incorporating a path integration module, which enables the agent to continuously update its internal representation of position. By comparing agents trained with and without explicit path integration, we find that path-integrating agents explore further away from origin and adjust their behavior in response to spatial uncertainty. These agents also develop clearer neural representations of past and future locations, supporting the emergence of internal maps — an ability fundamental to biological navigation. Furthermore, we investigate whether sparsely connected networks can achieve the same or better performance as fully connected ones, offering insights into the efficiency of biologically inspired neural architectures.

We have aslo developed two continuous attractor network modules in JAX that integrates velocity with almost no errors that can be implemented in any Flax Neural network architecture. The first is a numerical integrator that mirrors key biophysical features of entorhinal grid cells and can co-train with large brain-like networks. THe second is a lightweight analytical integration — ideal when velocity inputs are explicit — that updates the activity bump in closed form. Together they let researchers choose between biological richness and computational economy, and, to out knowledge, they constitute the first open-source, JAX-native toolkit for adding grid-cell dynamics to deep networks. In this thesis, both modules have been implemented in architectures, but only the analytical CAN has been tested extensively for computational complexity.

To substantiate the novelty and relevance of this work, Appendix A and B contain a poster and presentation from a seminar at the Lyon Neuroscience Research Centre [22], while Appendix C includes an abstract submitted to COSYNE 2025. Additionally, parts of this research has been submitted as a pre-print to NeurIPS; you can view it via the link provided [23].

1.3 Relation to the TMA4500 Project Report

This master's thesis expands upon the project report I completed for the course *TMA4500 – Industrial Mathematics* [19]. Consequently, certain material is reused, other parts are revised, and several sections are entirely new. To maintain transparency, each chapter (from now on) that draws on the project report begins with a brief note indicating which subsections originate from that previous work.

From the introduction, sections 1.1, 1.2, 1.4, 1.5 are adapted from [19] and retain many similarities with the original, despite being revised for this thesis.

1.4 Acknowledgements

I would like to express my gratitude to the Rajan Lab at Harvard Medical School for the opportunity to contribute to this project. In particular, I thank Ryan Badman and Riley Simmons-Edler for their guidance and for including me in their work. I appreciate their support and insights, which have greatly enriched my understanding of memory-driven reinforcement learning.

I am also grateful to my supervisor, Associate Professor Benjamin A. Dunn. His expertise in grid-cell research and computational neuroscience set the intellectual foundation for this thesis, and his steady encouragement gave me the confidence to tackle such an ambitious topic.

ChatGPT was actively used to refine and rephrase text for improved readability. Additionally, ChatGPT assisted in writing the code.

1.5 Structure of the thesis

This thesis is divided into two main parts:

Part 1 — Designing and Simulation the Reinforcement Learning System (Sections 2 - 5). This part introduces the foundations for the simulations, and then presents the results of the simulations. **Section 2** introduces the theoretical foundations of reinforcement learning and derives Proximal Policy Optimization (PPO) from first principles. **Section 3** introduces grid cells and presents continuous attractor models as a way of implementing them in a neural network. **Section 4** details the implementation of PPO in our specific setup, including the environment design, neural network architecture, and algorithmic framework. **Section 5** presents the simulation results and evaluates the performance of different network architectures.

Part 2 — Statistical Analysis of Space Representation (Sections 6 - 7). This part introduces methods to analyses the memory of the agents that was simulated in part 1, and then presents the results of the memory analysis. **Section 6** introduces the theoretical background of Generalized Linear Models, neural decoding, and techniques to analyse the continuous attractor networks. Section 7 presents experimental results, including behavioral decision modeling, neural decoding, grid-cell usage, and explores how spatial information is encoded and utilized in the agent's memory.

Finally, **section 8** and **section 9** summarize the findings, discuss key insights, and outline potential directions for future research.

1.6 Contribution to Sustainability

Although this project is primarily motivated by questions in computational neuroscience, two practical benefits could have modest sustainability value. These relate to the *United Nations Sustainable Development Goals* (SDGs), which identify key areas for global progress such as health, infrastructure, and environmental protection:

Energy-efficient autonomy for search-and-rescue and delivery. Our agents maintain strong navigation performance even after 90 % weight pruning, so the policy could fit on low-power edge hardware instead of relying on cloud compute. Such lightweight controllers could extend the battery life of delivery robots or search-and-rescue drones, supporting *Sustainable Cities and Communities* (SDG 11) and *Life on Land* (SDG 15).

Guidance for neuro-prosthetic design. Insights from analysis on future architectures built from the ones in this thesis, may inform future prosthetic designs that encode spatial information, contributing to *Good Health and Well-being* (SDG 3).

Part I

Designing and Simulating the Reinforcement Learning System

2 Foundations of Reinforcement Learning and Proximal Policy Optimization

This section introduces key concepts from reinforcement learning (RL), including belief states and memory, and derives the Proximal Policy Optimization (PPO) algorithm from first principles. The resulting formulation will be used as the loss function in our implementation (section 4).

The content of this section is similar to the material originally presented in [19], with adaptations for this thesis.

2.1 Markov Decision Process

A Markov Decision Process (MDP) provides a mathematical framework for sequential decision-making, where an agent interacts with an environment over discrete time steps. At each time step t, the agent observes a state s_t, takes an action a_t, and receives a reward r_t as a consequence. Crucially, the next state s_{t+1} depends only on the current state s_t and action a_t, which is known as the Markov property. The Markov property ensures that all relevant information about the past is contained in the present state. Formally, a finite-horizon MDP is defined as $\mathcal{M} = \{\mu, \mathcal{S}, \mathcal{A}, \mathcal{P}, r, H\}$:

- The distribution over initial states $\mu : \mathcal{S} \to [0, 1]$.
- The state space $\mathcal{S}$. A state at time t is denoted $s_t \in \mathcal{S}$ and represent the environment and agent configuration.
- The action space $\mathcal{A}$. The action at time t is denoted $a_t \in \mathcal{A}$.
- The dynamics model $P : \mathcal{S} \times \mathcal{A} \to \Delta\mathcal{S}$. The probability of transitioning to state s' from state s via action a is defined as $P(s'|s, a)$.
- The reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. The reward at time t is denoted as $r_t = r(s_t, a_t)$, which an agent receives for being in s_t and taking action a_t.
- The time horizon $H \in \mathbb{N}$ describing the length of a trajectory; the number of time steps before termination.

For infinite-horizon MDPs we omit the time horizon H and use γ instead:

- The discount factor $\gamma \in [0, 1)$. Describes how much future rewards are valued relative to immediate rewards.

2.1.1 Partially Observable MDPs

In many real-world scenarios, agents do not have full access to the state space $\mathcal{S}$. Instead, they receive partial observations s_t^{obs}, while certain aspects of the environment remain unobserved (s_t^{unobs}). This leads to a Partially Observable Markov Decision Process (POMDP), where the Markov property no longer holds for the observed state alone, meaning that all relevant information about the past is not contained in s_t^{obs}.

To compensate for this, POMDP agents can maintain a belief state b_t, which incorporaes past observations in memory (m_t) to approximate the true environment state:

$$s_t = \{s_t^{obs}, s_t^{unobs}\}, \quad b_t = f(s_t^{obs}, m_{t-1}).$$

The belief representation allows agents to make more informed decisions despite partial observability.

We will return to this framework in section 4, where we define the elements of our specific problem setup.

2.2 Policy

A policy π defines the agent's decision-making strategy as a probabilistic mapping from states to actions, expressed as $\pi : \mathcal{S} \to \Delta\mathcal{A}$. At each timestep, the agent samples an action from the policy, denoted as $a_t \sim \pi(\cdot|s_t)$. For simple problems with small, discrete state-action spaces, optimal policies (policies that maximize future rewards) can often be derived using classical dynamic programming techniques such as policy iteration [24]. However, in high-dimensional or partially observable environments, policies are typically represented as parametric functions π_θ, such as neural networks, with learnable parameters θ that are iteratively optimized to improve decision-making.

In POMDPs, policies must account for incomplete state information. Depending on the approach, the policy can be conditioned solely on the observed portion of the state, $a_t \sim \pi(\cdot|s_t^{obs})$, or it can integrate memory from past observations, represented as $a_t \sim \pi(\cdot|s_t^{obs}, m_{t-1})$. A common method for incorporating memory into the policy is through Long Short-Term Memory neurons, which enable the agent to retain and utilize past information when making decisions.

2.2.1 Long Short Term Memory (LSTM)

LSTM neurons are specialized types of neurons capable of maintaining hidden states over time [8]. Unlike standard feed-forward neurons, which process information in isolation, LSTM neurons can capture temporal dependencies by selectively retaining or discarding information through gates: the input gate, forget gate, and output gate. This ability to manage memory allows the agent to utilize information from previous observations or actions to make more informed decisions.

Let L_x be the size of the input layer before the LSTM layer and let L be the size of the LSTM layer. At each timestep t, an LSTM neuron recieves three input vectors:

- The hidden state from the previous timestep, $h_{t-1} \in \mathbb{R}^L$, which stores short-term information.
- The cell state from the previous timestep, $C_{t-1} \in \mathbb{R}^L$, which acts as long-term memory.
- The current input, $x_t \in \mathbb{R}^{L_x}$.

From this information we can generate the forget gate f_t, input gate i_t and output gate o_t. A temporary cell state $\tilde{C}_t$ is also generated directly from this vector with the tanh activation function.

$$\begin{aligned} f_t &= \sigma\big(W_f\,[h_{t-1}; x_t] + b_f\big) \in \mathbb{R}^L, \\ i_t &= \sigma\big(W_i\,[h_{t-1}; x_t] + b_i\big) \in \mathbb{R}^L, \\ o_t &= \sigma\big(W_o\,[h_{t-1}; x_t] + b_o\big) \in \mathbb{R}^L, \\ \tilde{C}_t &= \tanh\big(W_c\,[h_{t-1}; x_t] + b_c\big) \in \mathbb{R}^L \end{aligned}$$

The forget gate determines how much of the previous cell state C_{t-1} should be retained or discarded. A value close to 1 preserves information, while a value close to 0 removes it.

The input gate (i_t) regulates how much new information from the candidate cell state $\tilde{C}_t$ should be added.

The cell state is then updated as a weighted combination of past and new information:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t,$$

where $\odot$ is element-wise multiplication. Finally, the output gate (o_t) determines how much of the updated memory should be passed forward. The new hidden state is computed as:

$$h_t = o_t \odot \tanh(C_t).$$

This hidden state h_t serves two functions:

1. It is passed to the next timestep, preserving temporal information.

2. It propagates through the network, influencing decision-making at the current step.

In the context of neural policies, h_t can often be interpreted as a belief state, encapsulating the agent's memory of past observations. This makes the policy dependent on the hidden state, such that actions are sampled as:

$$a_t \sim \pi_\theta(\cdot|h_t).$$

2.3 Defining the RL objective

In reinforcement learning, the objective is to maximize cumulative rewards over time. This is formally captured by the value function, which quantifies the expected future rewards an agent can obtain when starting from a given state s and following a policy π_θ. In the finite-horizon setting (which we will adopt in our derivations), the value function is defined as

$$V^{\pi_\theta}(s) = \mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{h=0}^{H-1} r_h|s_0 = s\right]. \tag{1}$$

Here, τ represents a trajectory (or episode) with length H consisting of a sequence of states, actions, and rewards:

$$\tau = \{s_0, a_0, r_1, s_1, a_1, ...\}.$$

Each trajectory is generated by the policy π_θ, which defines the probability distribution over actions given a state, and the transition dynamics $P(s_{t+1}|s_t, a_t)$. The probability of a trajectory under policy π_θ is:

$$\rho_\theta(\tau) = \mu(s_0)\prod_{t=0}^{H-1}\pi_\theta(a_t \mid s_t)P(s_{t+1} \mid s_t, a_t),$$

where $\mu(s_0)$ is the initial state distribution. The notation $\tau \sim \rho_\theta$ in Equation 1 indicates that trajectories are sampled according to this probability distribution, meaning that when taking expectations over τ, we are integrating over all possible trajectories weighted by their likelihood under ρ_θ.

The reinforcement learning objective is to find the policy-parameters θ that maximize the expected cumulative reward across all possible initial states states

$$J(\theta) = \mathbb{E}_{s_0\sim\mu}[V^{\pi_\theta}(s_0)],$$

where J is the objective function that we want to maximize. In other words, the goal is to optimize the policy π_θ so that the agent accumulates the highest possible rewards over time. This optimization forms the foundation for policy-based reinforcement learning algorithms.

2.4 Policy Gradient Methods

With the objective of reinforcement learning in mind, we now turn our attention to deriving increasingly effective reinforcement learning algorithms. Our goal is to build upon fundamental principles to arrive at the method ultimately used for implementation — Proximal Policy Optimization.

We start by defining the return of a finite-horizon trajectory as

$$R(\tau) = \sum_{h=0}^{H-1} r_h.$$

The central objective of policy gradient methods is to maximize the expected return. This can be expressed as optimizing the function

$$J(\theta) = \mathbb{E}_{s_0\sim\mu}[V^{\pi_\theta}(s_0)] = \mathbb{E}_{\tau\sim\rho_{\pi_\theta}}\left[\sum_{h=0}^{H-1} r(s_h, a_h)\right] = \mathbb{E}_{\tau\sim\rho_\theta}[R(\tau)] = \int_\tau d\tau\rho_\theta(\tau)R(\tau).$$

We can write the gradient of the objective function as

$$\nabla_\theta J(\theta) = \nabla_\theta \int_\tau d\tau \rho_\theta(\tau) R(\tau) = \int_\tau d\tau \frac{\nabla_\theta \rho_\theta(\tau)}{\rho_\theta(\tau)} R(\tau) \rho_\theta(\tau) = \int_\tau d\tau \nabla_\theta \log \rho_\theta(\tau) R(\tau) \rho_\theta(\tau) = \mathbb{E}_{\tau \sim \rho_\theta}[\nabla_\theta \log \rho_\theta(\tau) R(\tau)]. \tag{2}$$

Because $\log(\rho_\theta(\tau))$ includes terms independent of θ, we simplify using the fact that only the policy terms depend on θ. Specifically,

$$\nabla_\theta \log \rho_\theta(\tau) = \nabla_\theta \log(\mu(s_0)\pi_\theta(a_0|s_0)P(s_1|s_0,a_0)...) = \nabla_\theta(\log \mu(s_0) + \log \pi_\theta(a_0|s_0) + \log P(s_1|s_0,a_0) + ...),$$

and we remove the θ-constant terms:

$$\nabla_\theta \log \rho_\theta(\tau) = \sum_{h=0}^{H-1} \nabla_\theta \log \pi(a_h|s_h).$$

Thus, inserting the expression into Equation 2, the gradient becomes

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \rho_\theta} \left[\left(\sum_{h=0}^{H-1} \nabla_\theta \log \pi_\theta(a_h|s_h) \right) R(\tau) \right]. \tag{3}$$

Recalling the definition of return $R(\tau)$, we can write

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \rho_\theta} \left[\sum_{h=0}^{H-1} \nabla_\theta \log \pi_\theta(a_h|s_h) \sum_{t=0}^{H-1} r(s_t,a_t) \right] = \sum_{h=0}^{H-1} \mathbb{E}_{\tau \sim \rho_\theta} \left[\nabla_\theta \log \pi_\theta(a_h|s_h) \sum_{t=0}^{H-1} r(s_t,a_t) \right].$$

We split the reward sum at $t = h$:

$$\sum_{t=0}^{H-1} r(s_t,a_t) = \sum_{t=0}^{h-1} r(s_t,a_t) + \sum_{t=h}^{H-1} r(s_t,a_t).$$

So the gradient becomes

$$\nabla_\theta J(\theta) = \sum_{h=0}^{H-1} \mathbb{E}_{\tau \sim \rho_\theta} \left[\nabla_\theta \log \pi_\theta(a_h|s_h) \sum_{t=0}^{h-1} r(s_t,a_t) \right] + \sum_{h=0}^{H-1} \mathbb{E}_{\tau \sim \rho_\theta} \left[\nabla_\theta \log \pi_\theta(a_h|s_h) \sum_{t=h}^{H-1} r(s_t,a_t) \right]. \tag{4}$$

In the first term of Equation 4 we can take the inner sum out of the expectation and get

$$\begin{aligned}
&\sum_{h=0}^{H-1} \mathbb{E}_{s_h,a_h \sim \rho_\theta} \left[\nabla_\theta \log \pi_\theta(a_h|s_h) \sum_{t=0}^{h-1} r(s_t,a_t) \right] \\
&= \sum_{h=0}^{H-1} \sum_{t=0}^{h-1} \mathbb{E}_{s_h,a_h \sim \rho_\theta} \left[\nabla_\theta \log \pi_\theta(a_h|s_h) r(s_t,a_t) \right] \\
&= \sum_{h=0}^{H-1} \sum_{t=0}^{h-1} \mathbb{E}_{s_h \sim \rho_\theta} \left[\mathbb{E}_{a_h \sim \pi_\theta(\cdot|s_h)} \left[\nabla_\theta \log \pi_\theta(a_h|s_h) r(s_t,a_t) \right] \right] \\
&= \sum_{h=0}^{H-1} \sum_{t=0}^{h-1} \mathbb{E}_{s_h \sim \rho_\theta} \left[r(s_t,a_t) \mathbb{E}_{a_h \sim \pi_\theta(\cdot|s_h)} \left[\nabla_\theta \log \pi_\theta(a_h|s_h) \right] \right].
\end{aligned} \tag{5}$$

In the third line we used law of iterated expectations, and in the last line we used that $r(s_t,a_t)$ does not depend on a_h when $t < h$, so it can be treated as a constant.

The inner expectation simplifies to:

$$\mathbb{E}_{a_h \sim \pi_\theta(\cdot|s_h)}[\nabla_\theta \log \pi_\theta(a_h|s_h)] = \sum_{a_h} \pi(a_h|s_h) \nabla_\theta \log \pi_\theta(a_h|s_h) = \nabla_\theta \sum_{a_h} \pi_\theta(a_h|s_h) = \nabla_\theta 1 = 0.$$

Since the term is zero, the entire sum in Equation 5 vanishes. Thus, we can remove the left-hand sum from Equation 4 and we are left with the expression:

$$\nabla_\theta J(\theta) = \sum_{h=0}^{H-1} \mathbb{E}_{\tau\sim\rho_\theta} \left[\nabla_\theta \log \pi_\theta(a_h|s_h) \sum_{t=h}^{H-1} r(s_t, a_t) \right]. \tag{6}$$

Using the law of iterated expectation, this can be further written as

$$\nabla_\theta J(\theta) = \sum_{h=0}^{H-1} \mathbb{E}_{s_h,a_h\sim\rho_\theta} \left[\mathbb{E}_{\tau\sim\rho_\theta} \left[\nabla_\theta \log \pi_\theta(a_h|s_h) \sum_{t=h}^{H-1} r(s_t, a_t)|s_h, a_h \right] \right].$$

The factor $\nabla_\theta \log \pi_\theta(a_h|s_h)$ is a constant under the inner conditional expectation on s_h and a_h. We move this factor outside the inner expectation and get

$$\nabla_\theta J(\theta) = \sum_{h=0}^{H-1} \mathbb{E}_{s_h,a_h\sim\rho_\theta} \left[\nabla_\theta \log \pi_\theta(a_h|s_h) \mathbb{E}_{\tau\sim\rho_\theta} \left[\sum_{t=h}^{H-1} r(s_t, a_t)|s_h, a_h \right] \right].$$

This expression can be simplified even further by introducing the Q-function. It is closely related to the value function (Equation 1) and describes the expected future reward given a current state-action pair (instead of just state as the value function is). It is defined as

$$Q_h^\pi(s, a) = \mathbb{E}_{\tau\sim\rho_\theta} \left[\sum_{t=h}^{H-1} r(s_t, a_t) \middle| s_h = s, a_h = a \right].$$

Recognizing this, we arrive at

$$\nabla_\theta J(\theta) = \sum_{h=0}^{H-1} \mathbb{E}_{s_h,a_h\sim\rho_\theta} \left[\nabla_\theta \log \pi_\theta(a_h|s_h) Q_h^{\pi_\theta}(s_h, a_h) \right],$$

which is the same as

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\rho_\theta} \left[\sum_{h=0}^{H-1} \nabla_\theta \log \pi_\theta(a_h|s_h) Q_h^{\pi_\theta}(s_h, a_h) \right]. \tag{7}$$

We have derived a objective gradient that can be implemented into a policy gradient method as follows:

$$\theta^{k+1} = \theta^k + \alpha \nabla_\theta J(\theta) \Big|_{\theta=\theta^k}.$$

However, the variance in Equation 7 is high, so we should add a baseline to the expression.

2.5 PG with learned baseline

Generally speaking, gradient estimators with lower variances lead to more stable improvement when used for gradient ascent. High variance can make the policy updates noisy and unstable, slowing or even preventing learning from converging. We will use baselines to induce lower variance in our gradient.

First, we note that the expectation of $\nabla_\theta \log \pi_\theta(a_h|s_h)$ multiplied with a constant is zero:

$$\mathbb{E}_{a\sim\pi_\theta(a|s)}[\nabla_\theta \log \pi_\theta(a|s)c] = c\mathbb{E}_{a\sim\pi_\theta(a|s)}[\nabla_\theta \log \pi_\theta(a|s)]$$

$$= c \sum_a \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} = c\nabla_\theta \sum_a \pi_\theta(a|s) = c\nabla_\theta 1 = 0.$$

From this, we see that shifting $Q_h^{\pi_\theta}(s_h, a_h)$ from Equation 7 with a constant baseline to does not introduce bias. Specifically, for any constant c, we have:

$$\mathbb{E}_{a_h\sim\pi_\theta(a_h|s_h)}[\nabla_\theta \log \pi(a_h|s_h)(Q_h^{\pi_\theta}(s_h, a_h) - c)] = \mathbb{E}_{a_h\sim\pi_\theta(a_h|s_h)}[\nabla_\theta \log \pi(a_h|s_h) Q_h^{\pi_\theta}(s_h, a_h)]$$

This property allows us to introduce a more general, state-dependent baseline ($b_h(s_h)$) without affecting the expected gradient. Using this, we further expand the gradient expectation:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{h=0}^{H-1}\nabla_\theta \log \pi_\theta(a_h|s_h) Q_h^{\pi_\theta}(s_h,a_h)\right] \\
&= \sum_{h=0}^{H-1}\mathbb{E}_{s_h,a_h\sim\rho_\theta}\left[\nabla_\theta \log \pi_\theta(a_h|s_h) Q_h^{\pi_\theta}(s_h,a_h)\right] \\
&= \sum_{h=0}^{H-1}\mathbb{E}_{s_h\sim\rho_\theta}\left[E_{a_h\sim\pi(\cdot|s_h)}\left[\nabla_\theta \log \pi_\theta(a_h|s_h) Q_h^{\pi_\theta}(s_h,a_h)|s_h\right]\right] \\
&= \sum_{h=0}^{H-1}\mathbb{E}_{s_h\sim\rho_\theta}\left[E_{a_h\sim\pi(\cdot|s_h)}\left[\nabla_\theta \log \pi_\theta(a_h|s_h)(Q_h^{\pi_\theta}(s_h,a_h)-b_h(s_h))|s_h\right]\right] \\
&= \mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{h=0}^{H-1}\nabla_\theta \log \pi_\theta(a_h|s_h)(Q_h^{\pi_\theta}(s_h,a_h)-b_h(s_h))\right] \qquad (8)
\end{aligned}$$

In other words, we do not affect the bias of the gradient by adding a baseline term that only depend on the current timestep. It can only change the variance.

We choose $b_h(s_h) = V_h^{\pi_\theta}(s_h)$, and define the advantage as

$$A_h^{\pi_\theta}(s_h,a_h) = Q_h^{\pi_\theta}(s_h,a_h) - V_h^{\pi_\theta}(s_h).$$

The expected advantage in a state s_h is

$$E_{a_h\sim\pi_\theta(\cdot|s_h)}[A_h^{\pi_\theta}(s_h,a_h)] = E_{a_h\sim\pi_\theta(\cdot|s_h)}[Q_h^{\pi_\theta}(s_h,a_h)] - V_h^{\pi_\theta}(s_h) = 0,$$

so using this centers the gradient around zero, leading to more stable learning.

Inserting the advantage into Equation 8, the gradient of the objective function becomes

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{h=0}^{H-1}\nabla_\theta \log \pi_\theta(a_h|s_h) A_h^{\pi_\theta}(s_h,a_h)\right]. \qquad (9)$$

We now have a form of the gradient with reduced variance, and once again an iterative scheme would be:

$$\theta^{k+1} = \theta^k + \alpha\nabla_\theta J(\theta)\Big|_{\theta=\theta^k}.$$

2.6 Trust Region Formulation

We now shift our focus to an alternative formulation that does not explicitly involve ∇_θ. Our goal is to identify a simpler expression in terms of θ^k that we can directly maximize at each iteration. Another benefit of this is that we can combine multiple objectives into a loss function.

From the result obtained in the last section (Equation 9) we derive

$$\begin{aligned}
\nabla_\theta J(\theta)\Big|_{\theta=\theta^k} &= \mathbb{E}_{\tau\sim\rho_{\theta^k}}\left[\sum_{h=0}^{H-1}\nabla_\theta \log\pi_\theta(a_h|s_h)A_h^{\pi_{\theta^k}}(s_h,a_h)\right]\Big|_{\theta=\theta^k} \\
&= \sum_{h=0}^{H-1}\mathbb{E}_{a_h,s_h\sim\rho_{\theta^k}}\left[\nabla_\theta \log\pi_\theta(a_h|s_h)A_h^{\pi_{\theta^k}}(s_h,a_h)\right]\Big|_{\theta=\theta^k} \\
&= \sum_{h=0}^{H-1}\mathbb{E}_{s_h\sim\rho_{\theta^k}}\left[E_{a_h\sim\pi_\theta(\cdot|s_h)}\left[\nabla_\theta \log\pi_\theta(a_h|s_h)A_h^{\pi_{\theta^k}}(s_h,a_h)\right]\right]\Big|_{\theta=\theta^k} \\
&= \sum_{h=0}^{H-1}\mathbb{E}_{s_h\sim\rho_{\theta^k}}\left[\int da_h\pi_\theta(a_h|s_h)\nabla_\theta \log\pi_\theta(a_h|s_h)A_h^{\pi_{\theta^k}}(s_h,a_h)\right]\Big|_{\theta=\theta^k} \\
&= \sum_{h=0}^{H-1}\mathbb{E}_{s_h\sim\rho_{\theta^k}}\left[\int da_h\pi_\theta(a_h|s_h)\frac{\nabla_\theta\pi_\theta(a_h|s_h)}{\pi_\theta(a_h|s_h)}A_h^{\pi_{\theta^k}}(s_h,a_h)\right]\Big|_{\theta=\theta^k} \\
&= \sum_{h=0}^{H-1}\mathbb{E}_{s_h\sim\rho_{\theta^k}}\left[\int da_h\nabla_\theta\pi_\theta(a_h|s_h)A_h^{\pi_{\theta^k}}(s_h,a_h)\right]\Big|_{\theta=\theta^k} \\
&= \nabla_\theta\sum_{h=0}^{H-1}\mathbb{E}_{s_h\sim\rho_{\theta^k}}\left[\int da_h\pi_\theta(a_h|s_h)A_h^{\pi_{\theta^k}}(s_h,a_h)\right]\Big|_{\theta=\theta^k} \\
&= \nabla_\theta\sum_{h=0}^{H-1}\mathbb{E}_{s_h\sim\rho_{\theta^k}}\left[E_{a_h\sim\pi_\theta(\cdot|s_h)}\left[A_h^{\pi_{\theta^k}}(s_h,a_h)\right]\right]\Big|_{\theta=\theta^k} \\
&= \nabla_\theta\mathbb{E}_{s_1,\ldots,s_{H-1}\sim\rho_{\theta^k}}\left[\sum_{h=0}^{H-1}E_{a_h\sim\pi_\theta(\cdot|s_h)}\left[A_h^{\pi_{\theta^k}}(s_h,a_h)\right]\right]\Big|_{\theta=\theta^k}
\end{aligned}$$

Summarizing the above derivation we have:

$$\nabla_\theta J(\theta)\Big|_{\theta=\theta^k} = \nabla_\theta\mathbb{E}_{s_1,\ldots,s_{H-1}\sim\rho_{\theta^k}}\left[\sum_{h=0}^{H-1}E_{a_h\sim\pi_\theta(\cdot|s_h)}\left[A_h^{\pi_{\theta^k}}(s_h,a_h)\right]\right]\Big|_{\theta=\theta^k}.$$

From this we define a surrogate, first-order objective function

$$\tilde{J}(\theta) = \mathbb{E}_{s_1,\ldots,s_{H-1}\sim\rho_{\theta^k}}\left[\sum_{h=0}^{H-1}E_{a_h\sim\pi_\theta(\cdot|s_h)}\left[A_h^{\pi_{\theta^k}}(s_h,a_h)\right]\right],$$

which has the property that

$$\nabla_\theta J(\theta)\Big|_{\theta=\theta^k} = \nabla_\theta\tilde{J}(\theta)\Big|_{\theta=\theta^k}.$$

Hence, if we only want local first-order improvements around θ^k, we can replace $J(\theta)$ by $\tilde{J}(\theta)$ for the purpose of choosing a local ascent direction.

In other words, we aim to maximize $\tilde{J}(\theta)$ under the current policy π_{θ^k} while ensuring that the update θ remains sufficiently close to θ^k for the first-order approximation to remain valid. This leads to the iterative update rule:

$$\theta^{k+1} = \arg\max_\theta \mathbb{E}_{s_0,\ldots,s_{H-1}\sim\rho_{\pi_{\theta^k}}}\left[\sum_{h=0}^{H-1}\mathbb{E}_{a_h\sim\pi_\theta(\cdot|s_h)}\left[A^{\pi_{\theta^k}}(s_h,a_h)\right]\right], \tag{10}$$

where θ^{k+1} is relatively close to θ^k.

2.7 Importance sampling

The method above requires sampling advantages from a distribution $\pi_\theta(\cdot|s_h)$ where θ is an unknown parameter. We have to rewrite the expression in a way so that we can sample from θ^k instead. The inner expectation

of the advantage is an expectation over the unknown policy π_θ. In order to sample from a known distribution we use a principle called importance sampling:

$$\mathbb{E}_{a\sim\pi_\theta}\left[A(a)\right] = \int_a \pi_\theta(a)A(a)da = \int_a \pi_{\theta^k}(a)\frac{\pi_\theta(a)}{\pi_{\theta^k}(a)}A(a)da = \mathbb{E}_{a\sim\pi_{\theta^k}}\left[\frac{\pi_\theta(a)}{\pi_{\theta^k}(a)}A(a)\right].$$

Using this on the result from Equation 10 we get

$$\theta^{k+1} = \arg\max_\theta \mathbb{E}_{s_0,\dots,s_{H-1}\sim\rho_{\pi_{\theta^k}}}\left[\sum_{h=0}^{H-1}\mathbb{E}_{a_h\sim\pi_{\theta^k}(\cdot|s_h)}\left[\frac{\pi_\theta(a_h|s_h)}{\pi_{\theta^k}(a_h|s_h)}A^{\pi_{\theta^k}}(s_h,a_h)\right]\right].$$

The two expectations are drawn with respect to the same policy, and the inner expectation is conditioned on the outer expectation. We can use the law of total expectation to rewrite the iterative scheme into

$$\theta^{k+1} = \arg\max_\theta \mathbb{E}_{\tau\sim\rho_{\theta^k}}\left[\sum_{h=0}^{H-1}\frac{\pi_\theta(a_h|s_h)}{\pi_{\theta^k}(a_h|s_h)}A^{\pi_{\theta^k}}(s_h,a_h)\right],$$

which we write as

$$\theta^{k+1} = \arg\max_\theta \mathbb{E}_{\tau\sim\rho_{\theta^k}}\left[\sum_{h=0}^{H-1} r_h^k(\theta)A^{\pi_{\theta^k}}(s_h,a_h)\right], \quad r_h^k(\theta) = \frac{\pi_\theta(a_h|s_h)}{\pi_{\theta^k}(a_h|s_h)}. \tag{11}$$

2.8 Proximal Policy Optimization (PPO)

From the expression above it is clear to see that if θ and θ^k differ much, the fraction of the policies can become very large leading to unstable updates. In addition, as previously mentioned, for the first-order approximation of the objective function $\tilde{J}(\theta)$ to hold, the updates in θ must remain small. Therefore, it is necessary to find a way to constrain these updates so that the policies stay close to each other. In the clipped PPO, this is achieved by clipping the ratio as follows:

$$1-\varepsilon \leq r_t(\theta) \leq 1+\varepsilon.$$

Additionally, in the standard PPO literature formulation [25], the sum is incorporated into an expectation over time, leading to the following optimization objective:

$$\theta^{k+1} = \arg\max_\theta \mathbb{E}_t\left[\min\{r_t^k(\theta)A_t^{\pi_{\theta^k}}, \text{clip}(r_t^k(\theta),1-\varepsilon,1+\varepsilon)\}A_t^{\pi_{\theta^k}}\right].$$

Finally, instead of using an iterative approach, we reformulate this in terms of a single objective function. This allows us to combine multiples objectives (loss functions) during the backpropagation process within a neural network. We also use a hat over the advantage and expectation to indicate they are sampled versions:

$$J^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t\left[\min\{r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta),1-\varepsilon,1+\varepsilon)\}\hat{A}_t\right].$$

Up to this point, the choice between a finite- or infinite-horizon formulation has not been crucial, since the advantage estimation already incorporates the returns. However, moving forward, it becomes more important to explicitly select the γ-discounted returns appropriate for infinite-horizon settings.

2.9 General Advantage Estimator (GAE)

Let us start by introducing the **Bellman equation** for infinite-horizon MDPs which we will need for the derivation of the GAE:

The infinite-horizon Q-function is defined as

$$Q^{\pi_\theta}(s,a) = \mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{h=0}^{\infty}\gamma^h r(s_h,a_h)\Big| s_0=s, a_0=a\right].$$

We can write a recursive definition by:

$$\begin{aligned}
Q^{\pi_\theta}(s,a) &= \mathbb{E}_{\tau\sim\rho_\theta}\left[r(s_0,a_0)\middle| s_0=s, a_0=a\right] + \mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{h=1}^{\infty}\gamma^h r(s_h,a_h)\middle| s_0=s, a_0=a\right] \\
&= r(s,a) + \gamma\mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{h=1}^{\infty}\gamma^{h-1} r(s_h,a_h)\middle| s_0=s, a_0=a\right] \\
&= r(s,a) + \gamma\mathbb{E}_{s'\sim P(\cdot|s,a)}\left[\mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{h=0}^{\infty}\gamma^h r(s_{h+1},a_{h+1})\middle| s_1=s'\right]\right] \\
&= r(s,a) + \gamma\mathbb{E}_{s'\sim P(\cdot|s,a)}\left[V^{\pi_\theta}(s')\right].
\end{aligned}$$

Summarizing, the Bellman equation for an iterative process is defined as

$$Q^{\pi_\theta}(s_h,a_h) = r(s_h,a_h) + \gamma E_{s_{h+1}\sim P(\cdot|s_h,a_h)}\left[V^{\pi_\theta}(s_{h+1})\right]. \tag{12}$$

Now, following literature on the **General Advantage Estimator (GAE)**, [26], we define the one-step temporal-difference (TD) error as

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

where we have that

$$\mathbb{E}_{s_{t+1}}[\delta_t^V] = \mathbb{E}_{s_{t+1}}[\overbrace{r_t + \gamma V(s_{t+1})}^{\text{Bellman Equation}} - V(s_t)] = Q(s_t,a_t) - V(s_t) = A(s_t,a_t).$$

From the expectation of δ_t^V, we see that it serves as an γ-unbiased estimator of the advantage. However, direct use of δ_t^V often results in high variance, making learning unstable. To mitigate this, the GAE introduces an exponentially weighted aggregation of multiple TD errors, effectively balancing bias and variance.

We define $\hat{A}_t^{(k)}$ as the discounted sum of TD errors. Because the TD errors contains $V(s_{t+1})$ and $V(s_t)$, the discounted sum has a telescoping effect:

$$\hat{A}_t^{(k)} := \sum_{l=0}^{k-1}\gamma^l\,\delta_{t+l}^V \;=\; -V(s_t) \;+\; r_t \;+\; \gamma\, r_{t+1} \;+\; \cdots \;+\; \gamma^{k-1}\, r_{t+k-1} \;+\; \gamma^k\, V(s_{t+k}). \tag{14}$$

The GAE is then defined as:

$$\begin{aligned}
\hat{A}_{t(\gamma,\lambda)}^{\text{GAE}} &= (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda\hat{A}_t^{(2)} + \lambda^2\hat{A}_t^{(3)} + \dots\right) \\
&= (1-\lambda)\left(\delta_t^V + \lambda(\delta_t^V + \gamma\delta_{t+1}^V) + \lambda^2(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V) + \dots\right) \\
&= (1-\lambda)\left(\delta_t^V(1+\lambda+\lambda^2+\dots) + \gamma\delta_{t+1}^V(\lambda+\lambda^2+\lambda^3+\dots)\right. \\
&\qquad \left.+\gamma^2\delta_{t+2}^V(\lambda^2+\lambda^3+\lambda^4+\dots) + \dots\right) \\
&= (1-\lambda)\left(\delta_t^V\left(\frac{1}{1-\lambda}\right) + \gamma\delta_{t+1}^V\left(\frac{\lambda}{1-\lambda}\right) + \gamma^2\delta_{t+2}^V\left(\frac{\lambda^2}{1-\lambda}\right) + \dots\right) \\
&= \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l}^V.
\end{aligned}$$

This can be written iteratively as

$$\hat{A}_{t(\gamma,\lambda)}^{\text{GAE}} = \delta_t^V + \gamma\lambda\hat{A}_{t+1(\gamma,\lambda)}^{\text{GAE}},$$

which we solve for every update-step of our algorithm. Note that δ_t^V requires a value estimation $\hat{V}(s)$, so our model needs to have an estimate for the value function of the current state. For deep reinforcement learning this popularly achieved using an actor-critic style neural network. Implementation of this architecture will be discussed further in subsection 3.4.

While **GAE introduces bias** to the exponential weighting, it significantly reduces variance by smoothing noisy advantage estimates. This tradeoff is controlled by the parameter λ, where lower λ values increase

bias but reduce variance, and higher λ values decrease bias, but at the cost of higher variance. In deep reinforcement learning, this balance is crucial for stable training, and λ is often tuned empirically.

Using the GAE as the advantage function in PPO, we get the following objective function:

$$J^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min\{r_t(\theta)\hat{A}^{\text{GAE}}_{t(\gamma,\lambda)}, \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\}\hat{A}^{\text{GAE}}_{t(\gamma,\lambda)} \right]. \tag{13}$$

In Section 4, we will describe in detail how this objective function is implemented and used to iteratively update the neural network during training.

3 Grid Cells and Continuous Attractor Networks

This section introduces the biological and theoretical foundations of grid cells and describes how their dynamics can be modeled through continuous attractor networks (CANs). We begin by reviewing key properties of grid cells, then outline how these spatial codes can emerge from recurrent network dynamics, and derive a mathematical model for continuous attractor networks that is later implemented in our architecture.

3.1 Introduction to Grid Cells

Grid cells, discovered in 2005 by the Moser group, reside in the medial entorhinal cortex (MEC) and fire in multiple locations in a gid pattern [27]. Each grd cell has a charachteristic spacing and orientation to its firing grid and neighbouring grid cells often share spacing but offset their grids (different "phases"). This effectively provides a coordinate system for space. Grid cells are active across environments, suggesting they privide a universal metric. Crucially, grid cells respond to the movement of the animal even in the absence of landmarks — they are believed to arise from an internal path integration process. In the brain, path integration is supported by head direction cells and speed cells which likely provide the velocity input to the grid cell network.

Place cells, discivered by O'Keefe in 1971, are hippocampal neurons that fire when an animal is in a specific location in an environment [28]. Different place cells have different preffered locations (place fields), and collectively they form an internal map of the current environment. Place cells are influenced by external sensory cues (landmarks, odors, etc) and can remap when the environment changes contect (source). They are though to be critical for spatial memory and for associating memories with locations (episodic memory).

The hippocampus and entorhinal cortex are heavily interconnected. A prevalent theory is that place cells may derive some of their spacial tuning from grid cell inputs [29]. Grid cells in layer II of MEC project to the hippocampus, possibly providing a metric input to help update which place cells should be active as the animal moves. In this view, grid cells offer a rough coordinate, while place cells tie that coordinate to specific environmental feaures or contexts. Alternatively, one can think of place cells as reading a combination of multiple grid cell firing patterns — indeed, mathematically a set of grid fields with different phases can uniquely code a location (like a basis representation). On the other hand, place cells can influence grid cells by providing periodic sensory corrections: when the animal encounters a familiar landmark, hippocampal place cell activity can realign or reset the grid cell network to prevent accumulated drift. In rodents, disrupting MEC grid cells inpairs path integration ability, while hippocampal lesions impair place memory — highlighting that both are needed for robust navigation and memory.

The motivation for creating a neural network architecture inspired by this dance between place cells and grid cells are many. Firstly, biological similarity between the neural architectures is a condition for comparing neural recording from biological and artificial agents. Secondly, grid-like representations (specifically in continous attractor networks) have been demonstrated to enable more efficient and higher capacity encoding of spatial variables, compared to non-grid coding. [30]. Thirdly, it has also been shown that grid code allows for high capacity learning in general [31]. We aim to see if the implementation of grid cells can allow for better encoding of space, but in addition if the grid cells choose to encode information about an abstract space not related to the spatial position of the agent. The "place cells", and all other general memory will be modeled through LSTM neurons, while we will model the grid cells with continuous attractor networks.

3.2 Continuous Attractor Networks

For intuition, one can image that grid cells are arranged on a two-dimensional sheet, where each neuron is connected to others via synaptic weights. Neurons close to each other are excitatory coupled, while distant ones are inhibited. This connectivity pattern causes neighboring neurons to co-activate, while suppressing activity elsewhere — producing a localized "activity hump". We model each neuron as a node whose activation value contributes to this bump, and whose dynamics follows a differential equation that governs how the bump moves.

Continuous attractor networks (CANs) are a class of recurrent neural networks in which the stable states lie on a continuous manifold. In the case of grid cells, each stable state corresponds to the position of the activity

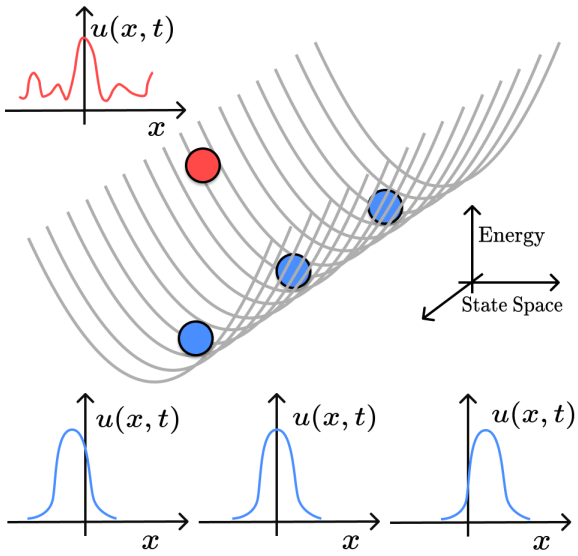


Figure 1: Intuitive 1D visualization of energy landscape of the continuous attractor network states. Any state that is not a stable bump (red) will relax into a lower-energy configuration (blue). Once formed, the bump can move smoothly accross the sheet along energy valleys.

bump on a neural sheet. Because this bump can shift smoothly across the sheet without changing shape or amplitude, the network effectively encodes a continuum of positions—making it well-suited for spatial representation. Figure 1 illustrates this concept in a 1D setting.

In the following subsections, we derive the governing equations of a CAN from first principles and show how this formalism can be implemented to model grid-cell-like dynamics.

3.3 RC Circuit Model of Neuron

The RC circuit is one of the simplest models for capturing the electrical dynamics of a neuron. It expresses how the membrane potential — the voltage difference between the intracellular and extracellular space — evolves in response to external inputs, using a first-order differential equation.

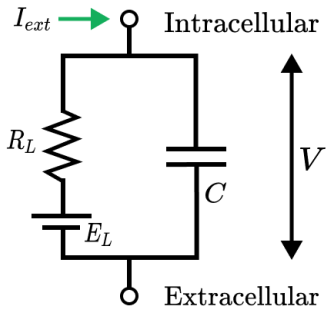


Figure 2: The RC circuit modelling the potenital difference between intracellular and extracellular space.

Both the intracellular and extracellular fluids contain freely moving ions and can be approximated as perfect conductors — i.e wires in the circuit analogy. The schematic in Figure 2 represents only the membrane; the rest of the cell is abstracted away. The simplification is sufficient to model how the membrane integrates incoming signals and generates spikes.

The lipid bilayer membrane itself acts as an insulator, but excess charge accumulates on both sides, allowing

it to be modeled as a capacitor C. A capacitor stores charge and builds up potential over time in response to current. However, the membrane is not a perfect insulator: it contains ion channels that allow specific ions to leak through, creating a passive current. This leakage is modeled as a resistor $R_L = 1/G_L$, where G_L is the membrane conductance.

The ionic composition on either side of the membrane is asymmetrical: potassium (K^+) is concentrated inside the cell, and sodium (Na^+) outside. This imbalance is actively maintained by the sodium-potassium pumps. Since the membrane is more permeable to K^+ than to Na^+, potassium tends to diffuse outwards more frequently, making the extracellular side more positive. The resulting voltage gradient stabilizes at the leak reversal potential $E_L \approx -70mV$. In the circuit model, this resting potential is represented by a battery in series with a resistor.

The capacitive current is defined as

$$I_c(t) = C\frac{dV}{dt},$$

where C is the membrane capacitance, which is a measurable physical constant for neurons.

The resistive current is given by Ohm's law as

$$I_R(t) = \frac{V(t) - E_L}{R},$$

where R is the membrane resistance and E_L is the leak reversal potential.

Applying Kirchoff's current law to the membrane, we equate the total current to the external input current $I_{ext}(t)$:

$$I_C(t) + I_R(t) = I_{ext}(t),$$

which gives

$$C\frac{dV}{dt} + \frac{V(t) - E_L}{R} = I_{ext}.$$

Multiplying by R, we define the external driving voltage $V_{ext}(t) = RI_{ext}(t)$, and note that the product RC has units of time. Letting $\tau = RC$, we obtain the canonical RC circuit equation for the membrane potential:

$$\tau\frac{dV}{dt} = -V + V_{ext}(t) - E_L. \tag{14}$$

This differential equation describes how the membrane voltage integrates incoming current over time, with exponential decay towards the driving potential and time constant τ.

In simplified models, a neuron is said to fire when the membrane potential reaches a threshold, typically around $V_{thresh} = -50mV$. This depolarization is driven by the rapid opening of voltage-fated sodium channels, which briefly makes the membrane potential positive. Subsequently, voltage-gated potassium channels open, repolarising — and often hyperpolarising — the membrane. This sequence is captured by the Leaky Integrate-and-Fire (LIF) model.

In the LIF model, the neuron integrates incoming current until $V(t)$ crosses the threshold. At that point, a spike is registered, and the potential is reset to a lower value, such as $V_{reset} = -75mV$, in the next timestep. The membrane then resumes integrating input from the reset value.

Using a typical membrane time constant $\tau = 10ms$, Figure 3 illustrates how the LIF neuron responds to various external stimuli.

Importantly, when the input current is sustained and sufficiently strong, the neuron fires repeatedly. This results in a firing rate that scales approximately with input amplitude—a key property exploited in many rate-based neuron models, which we will examine in more detail in the next subsection.

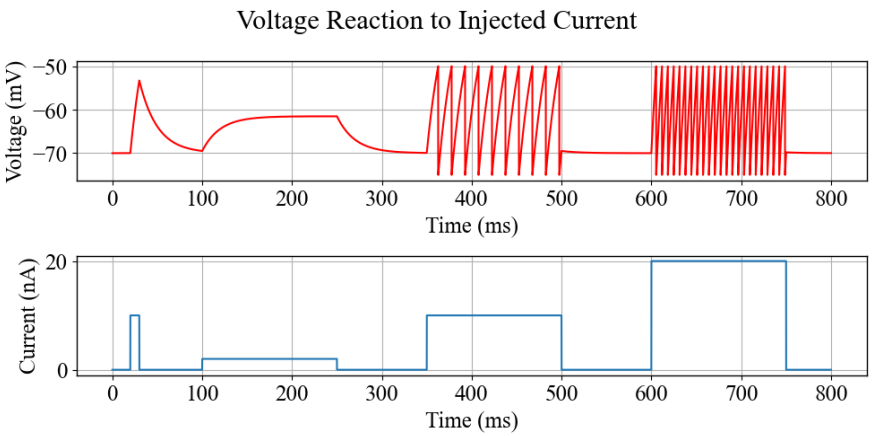


Figure 3: Voltage response of an RC circuit neuron to different input current profiles. Firing occurs when the membrane potential crosses threshold. The spike rate increases with input magnitude, reflecting proportional current-to-rate encoding.

3.4 Firing Rate Model

The leak reversal potential E_L shifts the baseline voltage but does not affect the firing rate itself. To simplify the analysis, we therefore set $E_L = 0$ and focus on the dynamics that determine firing frequency. With this assumption, the RC circuit equation from Equation 14 becomes

$$\tau \frac{dV}{dt} = -V + V_{ext}(t), \quad V_{ext}(t) = I_{ext}(t) R_L.$$

We now consider a constant positive input current ($V_{ext}(t) = V_{ext}$), as in the interval from $t = 350ms$ to $t = 500ms$ in Figure 3.

The solution to the differential equation in this regime is

$$V(t) = V_{ext} + (V(0) - V_{ext}) e^{-\frac{t}{\tau}}.$$

To analyse the dynamics between spikes, we assume $V(0) = V_{reset} = 0$. This simplifies the expression further and makes it easier to isolate the core relationship between input current and firing rate. These assumptions do not affect the general conclusions but help avoid clutter from constant offsets in the derivations.

We then obtain the membrane potential as a function of time:

$$V(t) = V_{ext}(1 - e^{-\frac{t}{\tau}}).$$

To cumpute the firing rate, we determine the time T it takes for the membrane potential to reach the threshold V_{thresh}:

$$V_{thresh} = V_{ext}(1 - e^{-\frac{T}{\tau}}).$$

We can solve this equation for T to obtain

$$T = \tau \ln \left(\frac{V_{ext}}{V_{ext} - V_{thresh}} \right).$$

The firing rate as a function of the external potential is then given by

$$r(I) = \frac{1}{T(I)} = \frac{1}{\tau \ln \left(\frac{V_{ext}}{V_{ext} - V_{thresh}} \right)}, \quad V_{ext} > V_{thresh}. \tag{15}$$

To better understand this expression, we consider the case where the input is much larger than the threshold ($V_{ext} >> V_{thresh}$). The fraction within the logarithm can be rewritten as

$$\frac{V_{ext}}{V_{ext} - V_{thresh}} = \frac{1}{1 - V_{thresh}/V_{ext}},$$

where we have that V_{thresh}/V_{ext} is very small. Using the Taylor expansion

$$\frac{1}{1-x} \approx 1 + x$$

for small x we get

$$\frac{1}{1 - V_{thresh}/V_{ext}} = 1 + \frac{V_{thresh}}{V_{ext}}.$$

Applying the expansion $\ln(1 + x) \approx x$ for small x, the firing period becomes

$$T(V_{ext}) = \tau \ln\left(1 + \frac{V_{thresh}}{V_{ext}}\right) \approx \tau \frac{V_{thresh}}{V_{ext}}.$$

Thus, the firing rate simplifies to

$$r(V_{ext}) = \frac{V_{ext}}{\tau V_{thresh}},$$

revealing the nearly linear relationship between input current and firing rate for strong inputs. This proportionality underpins the use of LIF neurons in many rate-based network models.

If we plot the firing rate expression from Equation 15 using a threshold $V_{thresh} = 20mV$, we obtain the curve shown in Figure 4.

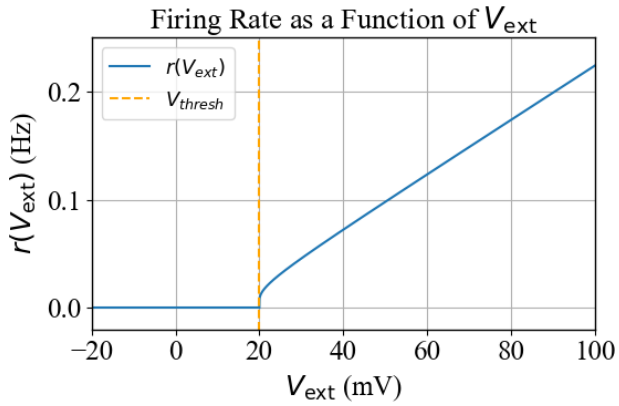


Figure 4: Firing rate of a postsynaptic neuron as a function of injected current. The relationship becomes approximately linear for sufficiently large inputs. This behaviour motivates the use of ReLU as a biologically inspired activation function.

As the plot reveals, the neuron does not fire when $V_{ext} < V_{thresh}$, and begins firing at an approximately linear rate once $V_{ext} > V_{thresh}$. This nonlinear, thresholded behaviour is qualitatively similar to the ReLU (Rectified Linear Unit) activation function commonly used in artificial neural networks. The ReLU is defined as $\text{ReLU}(x) = \max(0, x)$.

Thus, we see that the firing rate of a leaky integrate-and-fire neuron increases roughly linearly with the strength of external input — provided the input exceeds threshold. This observation justifies the use of the ReLU as a simple model of neuronal activation in the rate-based neural networks.

3.4.1 Neural Network Model

Suppose we consider a simple system with two neurons, where neuron v sends signals to neuron u through a synapse with weight w. The synaptic current into neuron u can be approximated by the product of the presynaptic firing rate and the synaptic weight:

$$I_{syn}(t) = wv(t).$$

The firing rate of the postsynaptic neuron is then modeled as

$$u = F[I_{syn}] = F[wv],$$

where F is the neuronal activation function. As shown in the previous section F can be approximated by the ReLU function. While the firing rates u and v are non-negative, the synaptic weight w can be either positive (excitatory) or negative (inhibitory). The simple system is illustrated in Figure 5

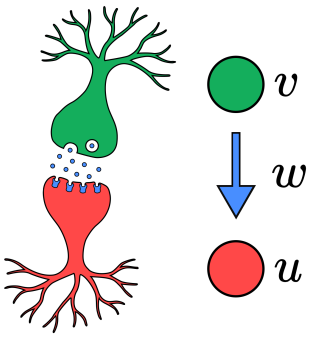


Figure 5: Illustration of the relationship between biological neurons and a firing-rate model.

To account for spatial integration from multiple presynaptic neurons, the firing rate of a single neuron is generalized to

$$u = F\left[\sum_b w_b v_b\right].$$

For a population of neurons we therefore have

$$\vec{u} = F[W\vec{v}],$$

where W is the weight matrix, $\vec{v}$ are the presynaptic firing rates and $\vec{u}$ are the postsynaptic firing rates. This represents the steady-state firing rate of the system — the output when the inputs has been constant for some time: $d\vec{u}/dt = 0$. To model the temporal evolution of activity, we introduce the dynamics described in the RC circuit model of neurons:

$$\tau\frac{d\vec{u}}{dt} = -\vec{u} + F[W\vec{v}].$$

To endow the neurons with memory, we introduce recurrent connections by adding input from $\vec{u}$ to itself via a matrix M. We also allow for a constant external drive $\vec{i}_{ext}$ representing inputs from other systems. This leads to the core dynamical equation:

$$\boxed{\tau\frac{d\vec{u}}{dt} = -\vec{u} + F[W\vec{v} + M\vec{u} + \vec{i}_{ext}]} \tag{16}$$

The input vector $\vec{v}$ can, in principle, represent any signal and be of arbitrary dimensionality. In this thesis, we focus on the specific case where $\vec{v} \in \mathbb{R}^2$ represents a two-dimensional velocity vector. The matrix W then acts as a mapping from velocity to a velocity-dependent activation pattern — or velocity-spike field — over the grid cell population. We denote this transformation as $\vec{h} = W\vec{v}$. The differential equation Equation 16 now describes how the grid cells integrate this velocity signal $\vec{h}$ over time, effectively implementing path integration.

In the next couple of sections, we will define and derive suitable forms for the vectors and the matrices in the framework to obtain a well-behaved continuous attractor network. Specifically, we will treat $\vec{u}$ as the activity of grid cells, construct the recurrent weight matrix M that supports stable integration, define a velocity-to-activation mapping W, and specify the role of external input $\vec{i}_{ext}$. Together these components will form the foundation for our continuous attractor model that we will later implement.

3.5 Defining the neural grid $\vec{u}$

The vector $\vec{u}$ represents neural activity over a two-dimensional sheet, often referred to as the neural grid, where the population of activity lives and moves. This sheet must have periodic boundary conditions: when

the bump exits one edge, it must re-enter from another edge. This ensures that movement across the sheet can be continuous and unbounded, even though the domain is finite.

To replicate the characteristic hexagonal firing patterns observed in grid cells, the geometry and topology of the neural grid must be chosen carefully. Specifically, the layout must support tiling patterns that preserve symmetry and local structure under periodic translation.

We model the neural sheet as a two-dimensional shape that tessellates an abstract space — often corresponding to physical position. There are at least three distinct ways to tile hexagonal firing fields into repeating units of equal area. These tiling options are illustrated in Figure 6.

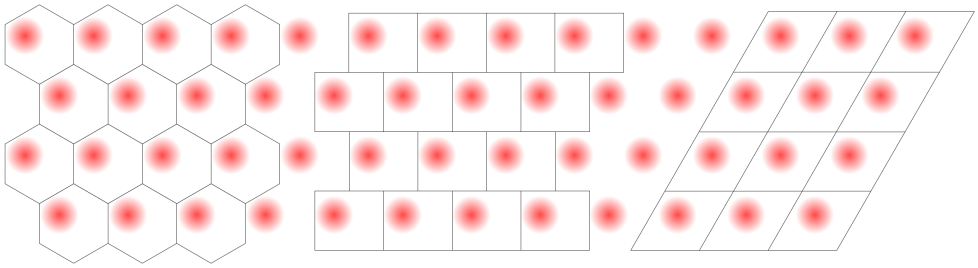


Figure 6: Three ways (known to us) to tessellate hexagonal patterns into repeatable tiles.

Each tessellation tile can be thought of as a fundamental domain through which the activity bump moves. As the bump crosses and edge, it reappears on the "opposite" side of the same tile — effectively wrapping around space. These wrapping rules define the boundary conditions of the domain, shown as colored edges in the left column of Figure 7.

To visualize the topology, we imagine gluing the colored edges together, forming a closed surface. The resulting topological object in all three cases is a twisted torus.

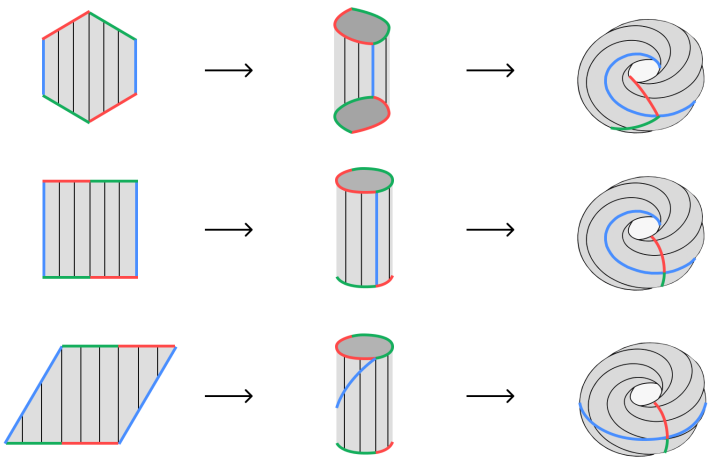


Figure 7: All three tessellations fold into a topological torus, but differ in how their boundary conditions align.

For this thesis, we choose the rhombic tessellation, as it yields simpler and untwisted boundary conditions when mapped to a torus. This makes it easier to define a regular index space for the grid-cell population, and simplifies both the implementation and the derivation of translation-invariant dynamics.

We can now define a grid of neurons placed within the rhombic domain. These neurons are arranged in an $N \times N$ lattice, where each neuron is indexed by integer coordinates $i_x, i_y \in \{0, 1, ..., N-1\}$.

Each neuron's location in the physical (or "neuron") space is determined by mapping its index to a continuous position within the rhombus. We define the position vector of neuron c_i as

$$\underbrace{c_i}_{\text{vector space}} = \underbrace{(c_{i_x}, c_{i_y})}_{\text{index space}} = \underbrace{\left(\frac{i_x + 0.5 i_y + 0.75}{N}, \frac{\sqrt{3}}{2} \cdot \frac{i_y + 0.5}{N} \right)}_{\text{neuron space}}.$$

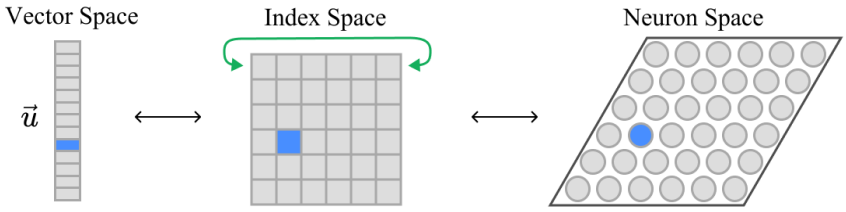


Figure 8: Relation between vector space, index space and neuron space representation of a grid cell.

This mapping preserves the regular triangular lattice structure necessary for generating hexagonal firing fields. It also respects the rhombus's periodic boundary conditions. For example, moving horizontally from $c(i, N-1)$ wraps around to $c(i, 0)$ as visualized by the green arrow in the "Index Space".

The population vector $\vec{u} \in \mathbb{R}^{N^2}$ thus corresponds to a structured sheet of neurons, each with a well-defined position in 2D space. This structure allows us to define spatial interactions — such as synaptic connectivity — based geometric distance between neurons.

3.6 Direction preference and Velocity Field $W\vec{v}$

To use the CAN as a path integrator, the activity bump must translate across the neural sheet in response to the animal's movement. This means that the velocity of the animal — assumed to be represented by a vector $\vec{v} = (v_x, v_y)$ — must influence the neural dynamics in a way that reliably shifts the bump in the direction of motion.

A common modeling approach is to include a feedforward input that selectively excites neurons aligned with the current velocity direction. This idea was first popularized by Fuhs and Touretzky [32], who proposed that neurons could receive direction-specific inputs and project asymmetrically offset recurrent weights. The effect is intuitive: when the animal moves northeast, neurons tuned to northeast directions become more active and help push the bump in that direction on the neural sheet.

If the bump translates proportionally to the physical movement, each neuron's activation will cycle on and off as the bump enters and exits its location in neural space. Over time, this generates multiple firing fields that tile physical space in a hexagonal pattern — mirroring the classic grid cell signature.

In the dynamic from Equation 16, the velocity input $\vec{v} \in \mathbb{R}^2$ is broadcast across the neural population using a matrix $W \in \mathbb{R}^{N^2 \times 2}$, transforming the low-dimensional velocity into a high-dimensional velocity-modulated drive:

$$\vec{h} = W\vec{v} \in \mathbb{R}^{N^2}.$$

To construct this mapping, we assign each neurons with a preferred direction $\hat{e}_{\theta_i}$ — a unit vector pointing in some angle θ_i. The contribution of the velocity input to neuron i's activation is then

$$h_i = \alpha \hat{e}_{\theta_i} \cdot v,$$

where α is a scaling factor controlling the strength of the modulation. When $\vec{v}$ aligns with a neuron's preferred direction, the dot product is maximized, and the neurons receives an enhanced excitatory input. This localized overactivation helps "pull" the activity bump in the corresponding direction.

Importantly, when there is no movement ($\vec{v} = 0$), the dot product vanishes, and the velocity-induced bias disappears. To maintain a bump amplitude in the absence of movement, a baseline input is typically added to the external drive. Figure 9 visualizes the direction preferences we use for our simulation.

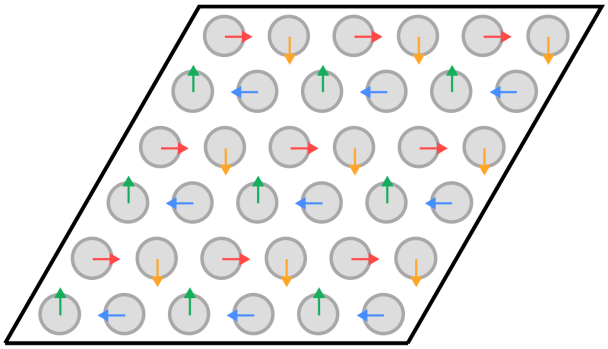


Figure 9: Each neuron is assigned a preferred direction. When the velocity input aligns with a neuron's preference, it receives stronger excitation and contributes to shifting the bump in that direction.

This mechanism provides a biologically plausible way to link velocity inputs to spatial transitions on the attractor sheet — allowing the network to integrate movement over time and generate the hallmark periodic grid pattern in physical space.

3.7 Distance on the Neural Sheet

To construct the recurrent weight matrix for a continuous attractor network, we need a consistent way to measure the distance between neurons on the neural sheet. However, because the sheet is topologically a twisted torus, distance is non-trivial.

3.7.1 Grid Norm

We define the distance between two neurons c_i and c_j using the following periodic metric:

$$\|c_i - c_j\|_{tri} = \min_k \|c_i - c_j + s_k\|,$$

where $s_k \in \mathbb{R}^2$ are shift vectors corresponding to wrapping around the torus in various directions. These shifts reflect the symmetry underlying the triangular lattice:

$$
\begin{aligned}
\mathbf{s}_1 &:= (0, 0), \\
\mathbf{s}_2 &:= \left(-0.5, \frac{\sqrt{3}}{2}\right), \\
\mathbf{s}_3 &:= \left(-0.5, -\frac{\sqrt{3}}{2}\right), \\
\mathbf{s}_4 &:= \left(0.5, \frac{\sqrt{3}}{2}\right), \\
\mathbf{s}_5 &:= \left(0.5, -\frac{\sqrt{3}}{2}\right), \\
\mathbf{s}_6 &:= (-1, 0), \\
\mathbf{s}_7 &:= (1, 0), \\
\mathbf{s}_8 &:= \left(-1.5, -\frac{\sqrt{3}}{2}\right), \\
\mathbf{s}_9 &:= \left(1.5, \frac{\sqrt{3}}{2}\right),
\end{aligned}
$$

This definition ensures that we always select the shortest wrapped distance on the torus, preserving continuity of the bump across periodic boundaries.

It's worth noting that the same metric would apply under the other two tessellations (hexagonal and square from subsection 3.5), though in those cases the seven core shifts s_1 through s_7 is sufficient, and s_8 and s_9 become redundant. In Figure 10, we illustrate how one would find the minimum distance between two neurons in the three different tessellations. We note that the shift vector used are not necessarily the same in the different tessellations.

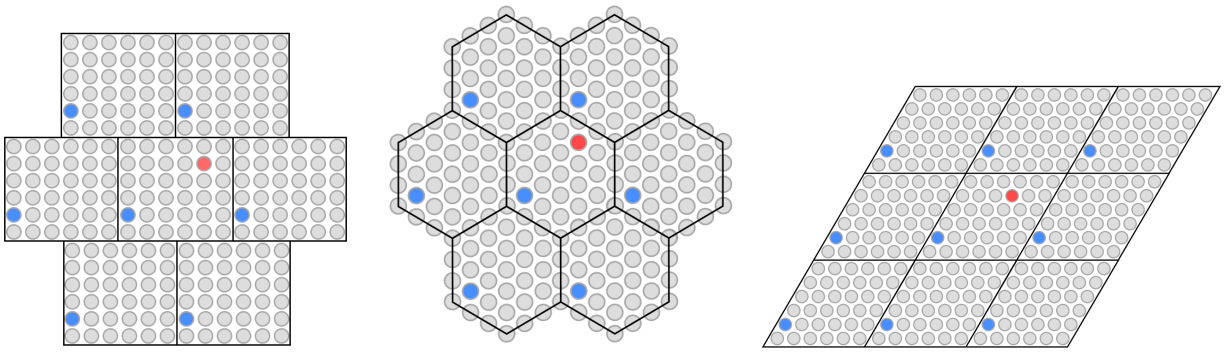


Figure 10: Comparison of coordinate systems and boundary wrapping across square, hexagonal, and rhombic tessellations. The blue points highlight how equivalent neurons are located under periodic wrapping.

3.7.2 Asymmetry in Distance Function

To incorporate directional modulation, we define an asymmetric distance function:

$$d_{tri}(c_i, c_j) = \|c_i - c_j - l\hat{e}_{\theta_j}\|_{tri},$$

where $\hat{e}_{\theta_j}$ is the preferred direction of neuron j, and l is a small displacement factor. This shifted distance effectively shortens the distance from c_i to c_j when c_j prefers movement in the direction of c_i, biasing the

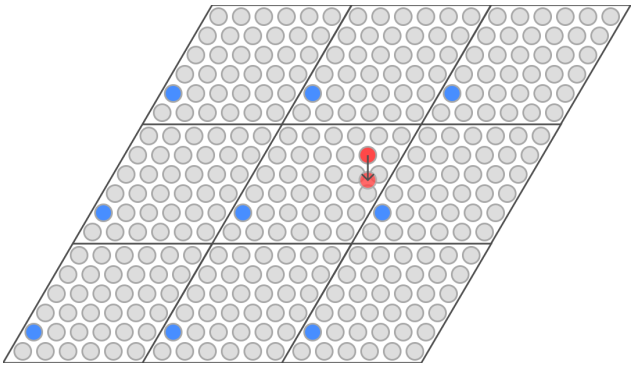


Figure 11: The distance between two points is the shortest way from the shifted neuron (red) to the wrapped versions of the other neurons (blue).

dynamics accordingly. In Figure 11 the distance between the blue and the red neurons have become shorter due to a shift in the red neuron before calculating the distance.

This mechanism allows direction-tuned neurons to influence bump motion in their preferred direction more strongly, which is essential for velocity-driven translation of the attractor.

This asymmetric distance function will be used to construct the recurrent weight matrix in the next section, encoding both spatial proximity and directionality—key features for stable and responsive bump dynamics.

3.8 Determine the Recurrent Matrix M

A central challenge in designing continuous attractor networks (CANs) is to ensure that the activity bump is both stable—maintaining its shape over time—and mobile, allowing it to move smoothly in response to velocity inputs. Achieving this balance requires careful tuning of the recurrent weight matrix M.

A classic design choice is the Mexican hat connectivity profile, where each neuron strongly excites nearby neurons and inhibits those farther away. This results in local positive feedback that sustains the bump, while inhibition at larger distances prevents the activity from spreading uncontrollably.

However, tuning M to support bump stability and accurate velocity integration is non-trivial. After testing several configurations, we arrived at the following formulation:

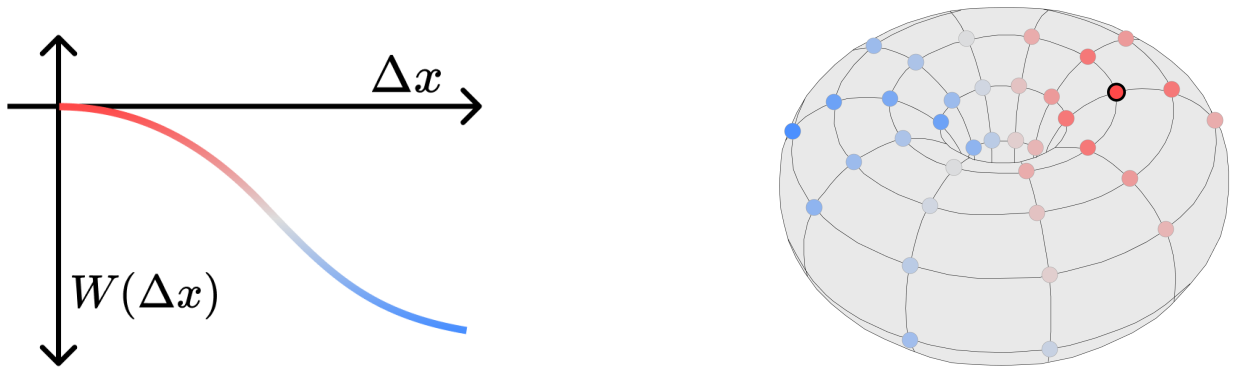
$$M_{ij} = A \exp\left(-\frac{\|x_i - x_j - l\hat{e}_{\theta_j}\|^2}{2\sigma^2}\right) - A. \tag{17}$$

Here, A controls the amplitude of the kernel, σ determines the spatial width, l is a shift in the preferred direction and $\|\cdot\|$ is the wrapped grid distance as previously defined.

This formulation produces a purely negative kernel for all nonzero distances. As such, the recurrent matrix on its own would not sustain a bump — activity would decay without an external drive. However, with the addition of a constant positive external input, the network integrates that input to maintain a stable bump, so we set $\vec{i}_{ext} = 1$.

This setup has several practical benefits. i) Nearby neurons still receive relatively less inhibition than distant ones, functionally mimicking the center-surround excitation-inhibition. ii) The purely negative recurrent weights avoid runaway positive feedback loops, which can cause bump amplitudes to diverge. iii) Bump magnitude becomes easier to stabilize and control.

In Figure 12, panel (a) shows the inhibitory kernel profile, and panel (b) visualizes how this profile applies across the 2D neuron sheet, producing localized activation surrounded by suppression. This carefully balanced recurrent matrix is critical for maintaining smooth and biologically plausible bump dynamics in our CAN model.



(a) (b)

Figure 12: (a) Profile of the recurrent weight function across distance, showing local disinhibition surrounded by stronger inhibition. (b) The effective excitation/inhibition profile distributed over the toroidal neural sheet.

3.9 Stability Analysis of the Equation

The goal of this section is to analyze the behavior of our dynamical system under different hyperparameter choices and understand the conditions under which the network maintains a stable and mobile activity bump. In particular, we are interested in identifying regimes where the bump:

- Maintains its shape (structural integrity)
- Translates smoothly in response to velocity inputs.

To facilitate the analysis, we first simplify the system by removing the nonlinearity introduced by the activation function $F(\cdot)$. This is justified by focusing out attention on the support of the bump, define at time t as

$$B_t = \{x : u_t > 0\}.$$

Within the active region, the firing rate $\vec{u}$ is positive, meaning the activation function behaves linearly. Thus, for neurons in the bump region, the system simplifies. At steady state ($d\vec{y}/dt = 0$), and without velocity input ($\vec{v} = 0$), the dynamical equation reduces to

$$\vec{u} = F(\vec{h} + M\vec{u} + \vec{i}_{ext}).$$

Since $\vec{u} > 0$ inside B_t, we can drop the activation function for the neurons in that region and we get

$$\vec{u} = \vec{h} + M\vec{u} + \vec{i}_{ext}.$$

This simplification generalizes to the moving bump case. For sufficiently small time steps Δt, the bump shifts only slightly between time steps and the support B_t changes minimally. Most neurons that were active remains active, allowing us to ignore the nonlinearity in this region and treat the dynamics as approximately linear.

Therefore, for the stability analysis that follows — focused on bump translation and maintenance — we restrict our attention to the interior of the bump and omit the activation function $F(\cdot)$ from the analysis.

Additionally, we assume $l = 0$ during the analysis, making the recurrent weight matrix M symmetric. This simplifies the derivations without qualitatively affecting the dynamics.

3.9.1 Stability of Eigenvalues

Under the simplifications introduced earlier — linearization within the bump region and symmetric M — the dynamics reduce to

$$\tau \frac{d\vec{u}}{dt} = -\vec{u} + \vec{h} + M\vec{u} + \vec{i}_{ext}.$$

Because M is symmetric we can diagonalize as

$$M\Phi = \Phi\Lambda,$$

where Φ is an orthonormal matrix of eigenvectors and Λ is a diagonal matrix of eigenvalues.

Substituting this into the system, we rewrite the dynamics as

$$\tau\frac{d\vec{u}}{dt} = -\vec{u} + \Phi\lambda\Phi^T\vec{u} + h + \vec{i}_{ext}.$$

Multiplying both sides by Φ^T transforms the system into the eigenbasis:

$$\tau\frac{d\Phi^T\vec{u}}{dt} = -\Phi^T\vec{u} + \Phi^T\Phi\lambda\Phi^T\vec{u} + \vec{h}_\Phi + \vec{i}_\Phi,$$

where we define $\vec{h}_\Phi = \Phi^T\vec{h}$ and $\vec{i}_\Phi = \Phi^T\vec{i}_{ext}$. Letting $\vec{c} = \Phi^T\vec{u}$, the dynamics become

$$\tau\frac{d\vec{c}}{dt} = -(I - \Lambda)\vec{c} + \vec{h}_\Phi + \vec{i}_\phi.$$

This equation is decoupled across eigenmodes. For each mode c_k, the solution is

$$c_k(t) = \begin{cases} c_k(0) + \frac{(h_\Phi + I_\Phi)_k}{\tau}t & \lambda_k = 1 \\ \frac{(h_\Phi + I_\phi)_k}{1-\lambda_k} + \left(c_k(0) - \frac{(h_\Phi + I_\phi)_k}{1-\lambda_k}\right)e^{-\frac{1-\lambda_k}{\tau}t} & \lambda_k \neq 1 \end{cases}$$

The solution reveals several important stability properties:

- **Unstable modes:** If any eigenvalue $\lambda_k > 1$, the corresponding mode grows exponentially with time, leading to divergence of the bump. Such recurrent matrices are rejected.
- **Stable integration:** Eigenvalues $\lambda_k \approx 1$ yield approximately linear growth over time, which is desirable for encoding continuous translation of the bump in response to velocity inputs.
- **Damped modes:** If $\lambda_k < 1$, the mode decays to a steady state value:

  $$c_k(\infty) = \frac{(h_\Phi + i_\Phi)_k}{1 - \lambda_k}.$$

  These modes respond to sustained inputs, but do not integrated over time.

When velocity is zero ($\vec{h} = 0$), we still require a steady state bump. This is achieved when $\vec{i}_{ext} > 0$ and eigenvalues as $\lambda_k < 1$.

Importantly, it turns out that six eigenvectors dominate the dynamics responsible for translating the bump across the sheet. These modes correspond to low-frequency, spatially coherent shifts in activity — aligned with the geometry of hexagonal grid tiling — and their eigenvalues are carefully tuned close to 1 when there's smooth movement without distortion.

3.9.2 Eigenmodes

To understand how activity bumps form and move within the recurrent network, we now analyze the eigenstructure of the recurrent matrix M. In the previous section, we established that bump stability and mobility depend on eigenvalues near one. Here, we explore the corresponding eigenvectors, which determine the spatial structure and symmetries of bump dynamics.

Assume that the neural sheet is an $N \times N$ rhombic lattice wrapped onto a torus, imposing periodic boundary conditions:

$$(i_x, i_y) \in \{0, \ldots, N-1\}^2.$$

The total number of neurons is $P = N^2$, and we write the activity vector $\vec{u}$ as

$$\vec{u} = (u_0, \ldots, u_{P-1})^\top \in \mathbb{R}^P.$$

With $\ell = 0$ the recurrent weight between neuron x_i and x_j depends **only** on distance $\|x_i - x_j\|$ (see Equation 17), and we can therefore write the elements of M as

$$M_{ij} = w(\vec{x}_i - \vec{x}_j) \,,$$

so for any i we have that

$$(M\vec{u})_i = \sum_{j=0}^{P-1} M_{ij} u_j = \sum_{j=0}^{P-1} w\big(\vec{x}_i - \vec{x}_j\big) u_j.$$

Writing $\vec{r} = \vec{x}_i - \vec{x}_j$ and using periodic boundaries turns the sum into a *discrete circular convolution*

$$(M\vec{u})(\vec{x}) \;=\; \sum_{\vec{r}} w(\vec{r})\, \vec{u}(\vec{x} - \vec{r}) \,.$$

To find the eigenvectors of M, we make an ansatz using the discrete fourier basis. We define the integer wave-vector $\vec{k} = (k_x, k_y)$ with $k_x, k_y \in \{0, \ldots, N-1\}$ and the corresponding Fourier mode as

$$\psi_{\vec{k}}(\vec{x}) \;=\; \frac{1}{\sqrt{P}}\, e^{\frac{2\pi i}{N}\, \vec{k}\cdot\vec{x}}, \qquad \vec{x} = (i_x, i_y).$$

These P functions are orthonormal and satisfy

$$(M\psi_{\vec{k}})(\vec{x}) = \sum_{\vec{y}} w(\vec{x} - \vec{y})\psi_{\vec{k}}(\vec{y}) = \psi_{\vec{k}}(\vec{x}) \sum_{\vec{r}} w(\vec{r}) e^{-2\pi i\, \vec{k}\cdot\vec{r}/N} = \widehat{w}(\vec{k})\psi_{\vec{k}}(\vec{x}).$$

This is summarized by writing it as an eigenpair as

$$M\psi_{\vec{k}} \;=\; \lambda(\vec{k})\, \psi_{\vec{k}},$$

with eigenvalue obtained by a discrete 2-D Fourier transform of the kernel

$$\lambda(\vec{k}) \;=\; \sum_{\vec{r}} w(\vec{r})\, e^{-\frac{2\pi i}{N}\, \vec{k}\cdot\vec{r}}.$$

This proves that the discrete Fourier modes diagonalize M, and the eigenvalues are the discrete Fourier transform of the weight kernel w.

Translation Modes:

The constant mode $\vec{k} = (0, 0)$ has eigenvalue

$$\lambda(0) = \sum_r w(r) < 0$$

because $w(\vec{r}) \leq 0$ everywhere.

The largest eigenvectors (responsible for integration of velocity) correspond to the six smallest non-zero wave-vectors, occurring for angles $0^o, 120^o$ and 240^o:

$$\vec{k}_n = k_0 \left(\cos\frac{2\pi n}{3}, \sin\frac{2\pi n}{3} \right), \quad n = 0, 1, 2.$$

They span a six-dimensional eigen-space, with a natural real-valued basis given by three cosine-sine pairs

$$\begin{aligned} \phi_n(\mathbf{x}) &= \cos\big(\tfrac{2\pi}{N}\mathbf{k}_n \cdot \mathbf{x}\big), \\ \psi_n(\mathbf{x}) &= \sin\big(\tfrac{2\pi}{N}\mathbf{k}_n \cdot \mathbf{x}\big), \qquad n = 1, 2, 3. \end{aligned}$$

The actual eigenvectors of M will generally be linear combinations of these basis functions:

$$u(\vec{x}) = \sum_{i=1}^{3} \alpha_i \phi_i(\vec{x}) + \beta_i \psi_i(\vec{x}).$$

Figure 13 shows both the individual modes ϕ_n and ψ_n (left) and a representative random linear combination that forms a bump-like pattern (right).

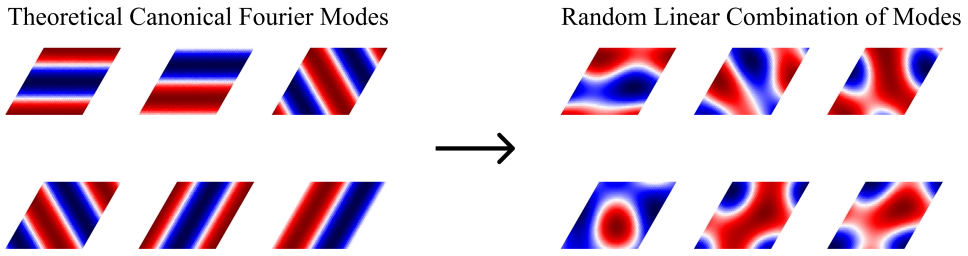


Figure 13: Theoretical fourier modes of the largest eigenvalue. On the left we see the simplest possible fourier modes, while on the right we see a random linear combination of the basis fourier modes.

We hypothesize that the six eigenvectors on the right are conceptually similar to the eigenvectors associated with the largest eigenvalues of the matrix M.

Visualizing Eigenpairs of M:

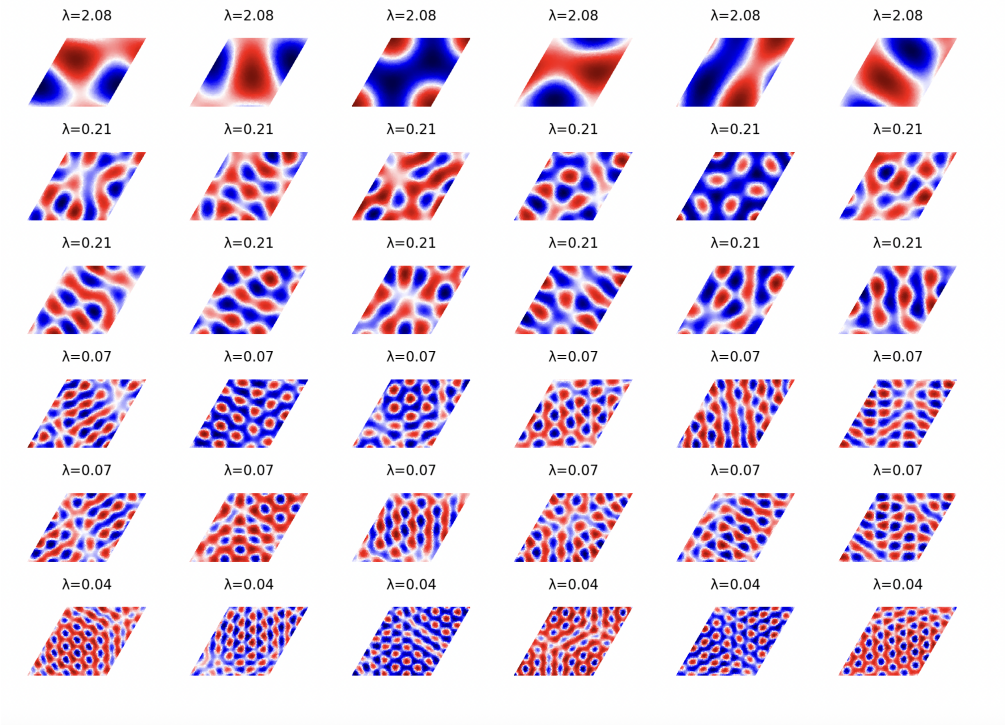


Figure 14: Top 36 eigen-pairs of the recurrent matrix. The first six (top row) correspond to translational modes.

Using a specific M matrix with carefully chosen parameters (discussed later), we can compute and visualize the top 36 eigenpairs. Figure 14 confirms that:

- The six largest eigenvalues correspond to low-frequency Fourier modes, resembling the "Random Linear Combination of Modes" generated theoretically.

- These six eigenvalues are the only ones exceeding 1, making them responsible for stable translation of the activity bump in response to velocity input.
- All remaining eigenvectors correspond to higher-frequency Fourier modes.
- Their eigenvalues satisfy $\lambda < 1$, meaning their response to constant input saturates to a fixed value. These modes shape the bump but to dot support persistent motion.

In summary, the six translation modes have eigenvalues slightly above one, enabling sustained integration of velocity signals. All other modes dampen over time and contribute only to the bumps stable shape. Notably, increasing the kernel width σ affects these smaller eigenvalues, thereby controlling the bump-width.

Lastly, while introducing $l > 0$ breaks strict symmetry, it does not alter the eigenvectors for large N. It only shifts eigenvalues downward by effectively centering the kernel asymmetrically. Thus, tuning l provides another mechanism to keep the translation eigenvalues close to one—crucial for stable integration.

We now turn to the choice of model parameters, where the primary objective is to ensure stable and accurate integration of velocity inputs.

3.10 Parameters

Choosing N. Burak and Fiete (2009) showed that finite networks introduce a "non-flat" energy landscape for the bump position: some positions are easier to move between than others, leading to variable step sizes and directional biases unless the network is sufficiently large [13]. This result imply that for stable and accurate path integration, the network's size N must be large enough to flatten the bump's energy landscape and minimize the position dependent distortions.

However, increasing N leads to quadratic growth in both memory and computation, since the neural activity vector $\vec{u} \in \mathbb{R}^{N^2}$. This makes training computationally expensive and increasingly impractical for network sizes around $N \approx 100$. In our implementation, we empirically found that maintaining accurate path integration over long episodes (up to 40,000 timesteps) required at least $N = 48$. This size offered a good trade-off: it ensured sufficiently smooth bump dynamics while keeping training time and memory usage within feasible bounds.

In Figure 15, we show how increasing N improves integration quality, with larger networks producing cleaner hexagonal firing patterns during a 40,000 timesteps random walks.

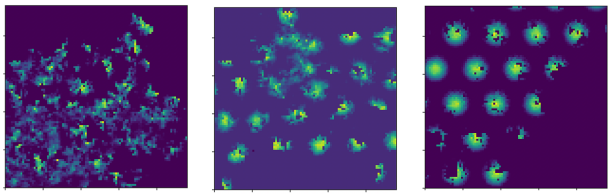


Figure 15: Firing patterns during random walk episodes for CANs with $N = 6$, 12, and 48. Higher N results in cleaner hexagonal patterns.

Choosing paramters of M. The weight matrix $M \in \mathbb{R}^{N^2 \times N^2}$ is governed by three parameters: the amplitude A, the offset length l and the kernel width σ. We fix $A = 1$ and explore how varying l and σ affects the largest eigenvalue λ_1 of M, summarized in Table 1.

We selected $\sigma = 200$ and $l = 6$ because they produced a stable, smoothly translating bump with a reasonable shape and amplitude when visualized. Notably, this parameter pair also aligns with the predictions from our earlier stability analysis, striking a balance where the largest eigenvalue remains above 1 but not excessively so.

When $\sigma = 300$, the bump failed to form entirely, while at $\sigma = 10$, the bump became so narrow that neighboring neuron activations were highly discontinuous — disrupting smooth integration.

$l \setminus \sigma$	10	50	100	200	300
0	241.42	31.46	8.22	2.08	0.93
1	240.04	31.28	8.17	2.07	0.92
3	229.21	29.86	7.80	1.97	0.88
6	195.07	25.41	6.64	**1.68**	0.75
12	91.67	11.95	3.12	0.79	0.35

Table 1: Largest eigenvalue λ_1 for each (l, σ) pair.

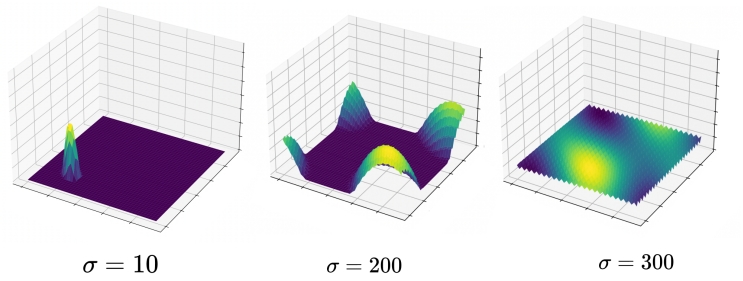


Figure 16: Bumps on the neural sheet for $\sigma = 10$, 200, and 300.

Empirically, maintaining $\lambda_{max} > 1$ appears necessary to preserve the bump integrity under continuous velocity inputs. Our chosen value of $\lambda_{max} = 1.68$ offers a robust compromise between integration accuracy and bump coherence.

Examples of the resulting bump profiles are shown in Figure 16.

Choosing Δt. The primary constraint when selecting the time-step Δt is to ensure numerical stability of the forward Euler iteration scheme. The stability condition requires that all eigenvalues of the Jacobian matrix $J = \frac{df}{du}$ lie within the method's stability region.

Given the system:

$$J = \frac{df}{du} = \frac{1}{\tau}[-I + M],$$

the eigenvalues of J are:

$$\mu_i = \frac{\lambda_i - 1}{\tau},$$

where λ_i are the eigenvalues of the recurrent weight matrix M.

Assuming the most negative eigenvalue of M is $\lambda_{min} = -9.19$ (which it is for the $\sigma = 200$, $l = 6$ setting), the most negative μ_i becomes:

$$\mu_{min} = \frac{-9.19 - 1}{\tau} = \frac{-10.19}{\tau}.$$

For forward Euler, the stability region requires

$$|1 + \Delta t \mu_i| < 1.$$

Applying this to the most negative μ_i gives

$$\Delta t < \frac{2}{|\mu_{min}|} = 0.196.$$

To ensure stability and reduce numerical integration error, we conservatively choose $\Delta t = 0.05$.

Choosing α . The parameter α scales the magnitude of the velocity-modulated input to neurons, and this determines the speed at which the activity bump translates across the neural sheet. To preserve the bump integrity, especially under maximal velocity inputs, α must be kept relatively small to avoid destabilizing the attractor. After trial and error, we set $\alpha = 0.1$, i.e., 10% of the baseline external drive $i_{ext} = 1$. This choice allowed the bump to move reliably without deforming or excessive amplitude fluctuations.

Multiple CANs. We instantiate multiple CANs operating in parallel, each with distinct frequency parameters. These frequency parameters can be trainable or fixed, and follow a geometric series. This is discussed further in subsection 4.4 about the neural network architecture.

The main constrain on the number of CANs is memory. For each CAN with $N = 48$, the recurrent matrix $M \in \mathbb{R}^{N^2 \times N^2}$ requires storing 48^4 float values. On a standard Slurm job (see subsubsection 4.6.3) with 16 CPUs and 8GB memory, this limits us to three CANs per node due to memory usage.

To validate the full setup, we conducted a long simulation episode with random walk velocity input, which was fed into all CANs. We tracked the firing activity of neuron (2,2) in each CAN and aggregated its actionvation over positions in the environment. The resulting activity map was plotted as a spatial heatmap (see Figure 17).

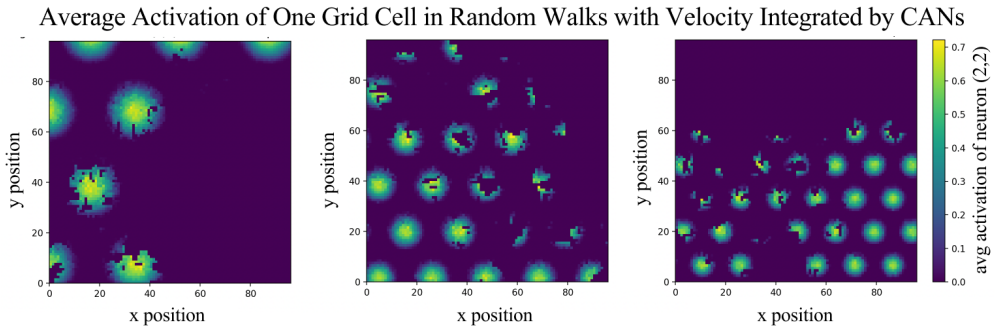


Figure 17: Accurate integration of velocity results in spherical circles in a hexagonal pattern.

The clear hexagonal structure in the firing fields confirms that each CAN reliably performs path integration. This demonstrates the viability of using multiple CAN modules in parallel to support rich, grid-like spatial encoding.

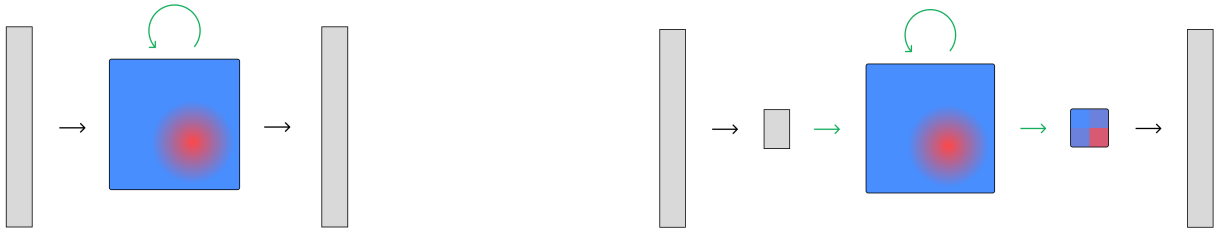
3.11 Implementation

Throughout our derivation, we used a low-dimensional velocity vector $v \in \mathbb{R}^2$ as input to the CAN. This input is expanded into the full neural space by using a velocity field matrix W, effectively shifting the bump in proportion to velocity. However, this is not the only feasible approach. An alternative would be to directly connect a high-dimensional vector from a preceding layer into the CAN, using dense all-to-all connections between the two layers as the new W. This would permit more complex and potentially richer dynamics, such as context-dependent modulation of the bump. However, it also dramatically increases the computational cost and memory usage, and is **left for future exploration**.

To make training efficient, we apply dimensionality reduction to both the input and the output of the CAN module. The CAN itself is represented as a 48×48 grid, and we denote the size of the adjacent network layer as L. Two architectural options are compared in Figure 18:

- In architecture (a), the full L-dimensional vector is connected densely to all $N^2 = 48^2$ neurons of the CAN, resulting in $2L \cdot 48^2 = 4608L$ trainable parameters.
- In architecture (b), the input is first compressed to a 2D velocity vector, and the output is projected back to a small spatial window (e.g. 2×2), requiring onlu $2L \cdot 2^2 = 8L$ parameters.

Reducing the dimensionality concentrate learning into fewer parameters, which ideally accelerate convergence and simplifies optimization.



(a) High-dimensional

(b) Low-dimensional

Figure 18: Comparison of architectures: (a) Full layer-to-grid connectivity; (b) Reduced input-output mapping via velocity encoding.

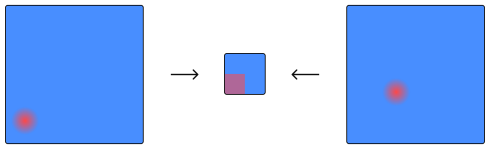


Figure 19: Preserving bump integrity under pixelation: as long as the bump is larger than the pixel grid, information is preserved. The image addresses the problem of having too few pixels.

While architecture (a) allows for more nuanced interactions—such as neurons that bias bump movement in response to environmental cues (e.g., detecting a predator)—the complexity and computational load make it a topic for future work.

We argue that the bump's translation, which is the core function of the CAN, can be equivalently achieved using a low-dimensional velocity vector as input, instead of a full L-dimensional embedding. Since the bump's steady-state shape is radially symmetric due to the distance-based recurrent weights, and since this shape is fully determined by the network parameters, the reduction does not lead to information loss—as long as the bump spans more than one pixel.

This architecture allows us to move and read out from the bump efficiently and with lower computational overhead.

Numerical Integration:

The neural dynamics are implemented using forward Euler integration. From the differential form:

$$\tau \frac{dv}{dt} = -v + F(h + Mv)$$

we approximate using discrete updates:

$$v^{t+1} = v^t + \frac{\Delta t}{\tau} \left[-v^t + F(Wx^t + Mv^t) \right].$$

Since the agent actions occur on a behavioral timescale (seconds), while neural dynamics evolve on a much finer timescale (milliseconds), we perform multiple integration steps for each velocity input. In practice we found that 50 Euler steps per velocity update offer a good balance between accuracy and efficiency.

3.12 Analytical Implementation for Computational Efficiency

While the previous sections derived and analyzed a full numerical implementation of a continuous attractor network (CAN), our final simulations adopt an analytical approach. This decision is motivated primarily by practical constraints—namely, memory usage, computational speed, and training duration.

As discussed, our model uses a learned velocity vector to drive the CAN bump dynamics. This simplification reduces the number of trainable parameters and enables easier interpretability, allowing us to correlate

velocity estimates with internal representations. It also makes future work—where richer, high-dimensional embeddings directly modulate bump movement—more tractable. As we will see in subsection 7.5, finding architectures that drive the bump with stable and meaningful velocity signals is a non-trivial challenge. Starting with this setup provided a useful foundation for understanding the design space and identifying promising directions for future work.

Given that we restrict the input to a velocity vector $\vec{v}_t \in \mathbb{R}^2$, bump translation can also be computed analytically, avoiding the need to simulate recurrent network dynamics at every timestep. This analytical formulation provides significant computational advantages while yielding behaviorally equivalent output to the numerical CAN when tuned appropriately.

Implementation Details:

Instead of evolving a high-dimensional state $\vec{u}_t$, we maintain a low-dimensional bump center $\vec{\mu}_t \in \mathbb{R}^2$, defined on a rhombic domain with periodic boundaries. This unit is bounded by

$$(0,0)\,, \quad (1,0)\,, \quad \left(1.5, \frac{\sqrt{3}}{2}\right), \quad \left(0.5, \frac{\sqrt{3}}{2}\right).$$

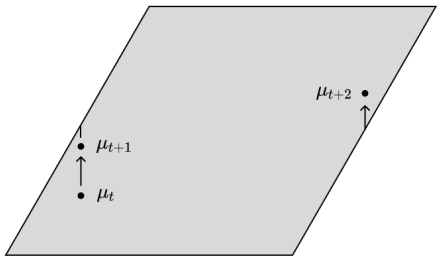


Figure 20: Movement of center of analytical bump with velocity in positive y-direction.

At each timestep, we update $\vec{\mu}_t$ using the learned velocity:

$$\mu_{t+1} = \mu_t + v_t \text{ mod rhombus.}$$

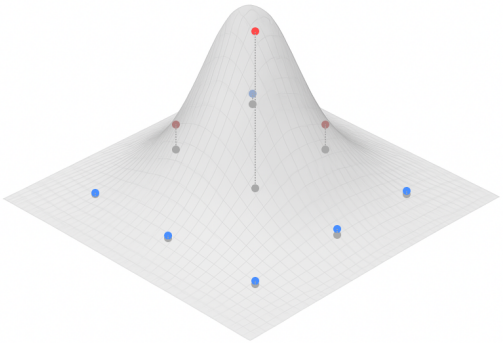


Figure 21: Sampling of analytical bump passed into the LSTM layer.

This preserves periodicity and mimics the effect of translation in the neural sheet. Figure 20 illustrates how the bump center moves with a constant velocity input in the positive y-direction.

We then generate a Gaussian bump centered at $\vec{\mu}_t$ and sample it on an $N \times N$ grid, producing the final CAN output. This sampled bump is passed into the downstream layers, as shown in Figure 21.

The bump's shape and speed are tunable to match the dynamics of the numerical CAN, allowing for direct comparisons. This analytical solution not only accelerates training, nut also dramatically reduces memory usage, allowing for multiple CANs in parallel — supporting experiments with richer or more hierarchical attractor structures.

4 Environment and Model Implementation

In this section, we implement the foraging task using the POMDP framework described in subsubsection 2.1.1, defining the observable states, action space, reward structure, and policy. We also present the loss function based on Proximal Policy Optimization (PPO) as introduced in section 2 and formalized in Equation 13, followed by the specific implementation of the algorithm for our model architecture.

This section builds on previous work presented in [19], and shares some similarities in structure and phrasing.

4.1 States (Environment)

The environment is a pseudo-randomly generated 2D grid world consisting of $n \times n$ tiles. Various objects populate the grid, each occupying a single tile. The key entities in the environment include a single agent, multiple predators, cows, water and other objects.

Entity	Health	Damage	Comments
Agent	10	varies (1–5)	Damage dealt depends on equipped weapon (e.g., sword)
Cow	3	0	Passive; can be consumed for food; does not attack
Zombie	5	2	Melee attacker; moves slowly; straightforward pursuit
Skeleton	3	2	Ranged attacker; fires arrows from up to 5 tiles away

Table 2: Health and damage stats for living entities in the environment.

The single **agent** has four parameters describing its well being:

- health h_t: Starts at 10 and decreases permanently when damaged. However, the agent can acquire a shield that would take damage instead of the agent itself.
- food f_t: Starts at 10 and describes how hungry the agent is. It decreases linearly over time and must be replenished by consuming cows.
- drink d_t: Similar to food, but is restored by drinking water (which is an easier task, because the agent does not need to kill the water.)
- energy e_t: Starts at 10 and and decreases linearly. The energy is restored by sleeping, which can only occur when $e_t < 5$ and the sleep continuous until $e_t = 10$.

There are two kinds of **predators** which both try to kill the agent. The zombie has 5 in health and deals 2 damage when adjecent to the agent. The skeleton has 3 in health and fire arrows from up to five tiles away, dealing 2 damage with a 70% probability.

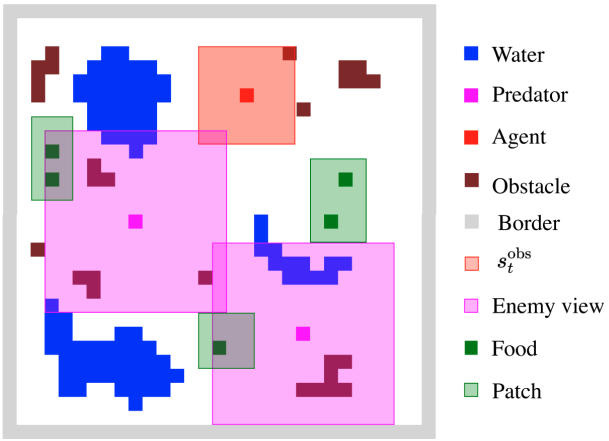


Figure 22: Design of environment. The size of arena, and the rectangles are not consistent with implementation, but it serves as a visualization for the reader to comprehend the various objects.

Cows have 3 health and spawn in designated, connected regions called patches. These patches are fixed features of the environment but are not directly observable by the agent. Once all cows in a patch are consumed, it becomes temporarily depleted, requiring time before new cows reappear. To forage effectively, the agent must learn to identify, leave, and revisit productive areas — a task that demands memory and long-term planning.

The environment also features **water** tiles, which are more likely to form connected structures such as rivers or lakes. **Obstacles** like stones and trees appear in clusters, forming forests or rocky areas. These can be harvested and stored by the agent as resources, enabling the crafting of tools and equipment such as axes, shields, and swords.

The full physical 2d grid state at time t, denoted as s_t, represents the entire environment and is encoded as a vector of length n^2. However, the agent does not have access to the full state; instead, it observes a local window s_t^{obs}, a rectangular region centered around the agent with dimensions $n_x^{\text{obs}} \times n_y^{\text{obs}}$. In addition to this partial observed state, the agent has access to its own inventory I_t and physiology ϕ_t. The inventory is stone, wood, iron, swords and so on, while the physeology is h_t, f_t, d_t, e_t.

This partial observability necessitates the maintenance of a **belief state** b_t, which represents the agent's internal estimate of the true state based on past observations.

A simplified model of the environment is illustrated in Figure 22.

4.2 Actions and dynamics

The set of actions $\mathcal{A}$ consists of 15 discrete actions a_t, including the actions *NOOP*, *LEFT*, *RIGHT*, *UP*, *DOWN* and *DO*. The first five actions are straightforward — for instance, *UP* moves the agent one tile upward unless obstructed by an obstacle. However, the outcome of the *DO* action is context-dependent, making the environment dynamics more complex.

For example, if the agent executes *DO* while next to a tree, it collects wood. This resource can later be used to craft items, such as a wooden sword via the *MAKE_WOOD_SWORD* action (action 14). The wooden sword, in turn, allows the agent to use *DO* defensively when in proximity to a predator, providing a means of self-protection. Attempting an action whose pre-conditions are unmet (e.g crafting without materials) results in a no-op.

Predators move to neighboring tiles, with their direction chosen uniformly when far from the agent, but heavily biased toward the agent when the agent is within the view of the predator ("Enemy view" in Figure 22). As a result, the environment exhibits stochastic dynamics, meaning that even for a given agent state and action, the outcome remains probabilistic due to the unpredictability of predator movements.

4.3 Reward

The agent's reward is tied to its physiological state, represented by the vector $\phi_t = (h_t, f_t, d_t, e_t)$, which captures health, food, thirst and energy levels. At each timestep, food, thirst, and energy levels naturally decline, requiring the agent to take proactive actions: drinking water replenishes thirst, consuming cows restores food and sleeping recovers energy. Health decreases only through attacks from predators.

The reward function encourages the agent to maintain homeostasis by assigning positive value to balanced internal states and penalizing neglect. Specifically, the reward at time t is

$$r_t = 0.1 \times (1 + \text{sign}(h_t - 5) + \text{sign}(f_t - 5) + \text{sign}(d_t - 5) + \text{sign}(e_t - 5))$$

This formulation grants increasing rewards as each parameter remains above a critical midpoint (value 5), and it imposes strong negative feedback if any parameter falls below threshold. If any of the physiological variables reach zero, the agent dies, ending the episode and triggering a large negative terminal reward.

This reward shaping ensures that the agent must first learn to satisfy its basic survival needs before progressing to higher-level behaviors like evading or defeating predators.

4.4 Policy Architecture

General Setup Without the CANs:

The policy is implemented as a recurrent neural network designed to emulate core components of biological decision making, as illustrated in Figure 23. The input to the network is a flattened version of the observed state s_t^{obs}, which is first passed through a fully connected layer of width L, followed by an LSTM layer also of width L.

At each timestep t, the LSTM receives the input vector x_t, along with the previous hidden state h_{t-1} and cell state C_{t-1}, and outputs the updated hidden state h_t. This hidden state serves as the input to the actor network, which consists of two fully connected layers of width L, culminating in a softmax output layer that defines a categorical policy over 15 discrete states.

The softmax operation is defined as:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{15} e^{z_j}} \quad \text{for } i = 1, \ldots, 15.$$

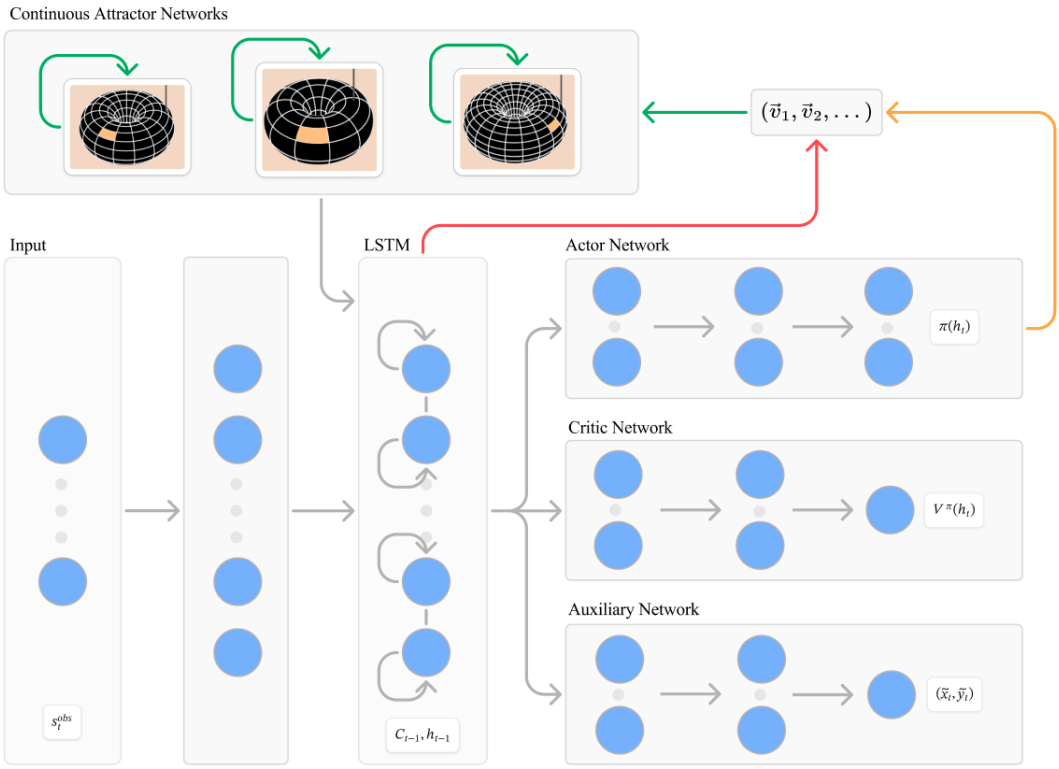


Figure 23: The architecture of the neural network. The actor head is used to determine policy, the critic head is used to estimate the GAE, and the auxiliary head is used for path-integrating agents. Green lines are fixed, grey lines are trainable, and red and orange lines are optional.

The policy is a function of the observed state, the memory, the hidden state and the recurrent state (if using the CANs): $a_t \sim \pi_\theta(\cdot|s_t^{obs}, C_{t-1}, h_{t-1}, r_{t-1})$. Alternatively it can be written as a function of the LSTM layer output as $a_t \sim \pi'_\theta(\cdot|h_t)$. Here h_t functions as the belief state b_t for POMDPs described in subsubsection 2.1.1.

The neural network architecture includes a critic network, which has the same structure as the actor network, consisting of two hidden layers. However, its output layer consists of a single neuron that estimates the value function $\hat{V}(h_t)$. This prediction is used in the calculation of the GAE in subsection 2.9 which is used in the PPO.

Additionally, we incorporate an auxiliary prediction network with an identical architecture to the critic network. This network predicts the agent's current position (x_t, y_t). Whether this network is included during training determines whether the agent employs path integration or not, distinguishing between a path-integrating and a non-path-integrating agent.

Additionally, we support training a sparse network using magnitude-based pruning after 20,000 epochs. The rationale is that weights with the smallest absolute values contribute minimally to the output and can be removed, reducing overparameterization and improving efficiency.

Adding the CANs to the Architecture:

The architecture also optionally integrates a grid cell module, as described in the previous section. This module receives velocity inputs and contributes recurrent state information r_t to the LSTM. Velocity input is computed in on of two ways, depending on the chosen architecture:

- **Architecture 1** derives velocity from the previous action taken.
- **Architecture 2** derives velocity from the LSTM's internal memory.

The orange line in Figure 23 corresponds to **Architecture 1**, which involves two matrix transformations. First, a fixed matrix maps the previous action a_{t-1} to a unit velocity vector $v \in (\pm 1, 0), (0, \pm 1)$ if the action involves translation ($a_{t-1} \in 1, 2, 3, 4$). Second, a trainable or predefined matrix transforms this unit velocity into m distinct velocity vectors, one for each of the m CAN modules.

In biological systems, particularly in rodents, grid cell modules have been observed to scale geometrically. Recordings in rats—and later in mice, bats, and humans—suggest that each module's spatial scale is roughly 1.4 to 1.5 times larger than the previous one, with all cells within a module sharing the same scale. Stensola et al. (2012) found a consistent scaling factor of 1.42 across modules [33], while Mathis et al. (2012) demonstrated that a scaling factor around 1.5 optimizes spatial decoding accuracy [34]. We adopt a scaling factor of 1.45, which falls within this biologically and functionally plausible range.

In the results section, we compare four variations of Architecture 1 based on different strategies for generating velocity scales:

- **Trainable scaling:** Each CAN has an independently trainable weight that adjusts the magnitude of its velocity input.
- **Trainable ratio:** Use the form $f_0 \cdot r^i$ with constant f_0 small enough to ensure that the coarsest module spans the entire arena, and trainable ratio r.
- **Trainable base:** Use the same geometric form, but with constant $r = 1.5$ and trainable base frequency f_0.

These configurations allow us to explore the impact of scale encoding mechanisms on the agent's spatial learning and representation.

The red line in Figure 23 represents **architecture 2**. It is a matrix with size $\mathbb{R}^{512 \times 2 \cdot m}$, where m is the number of CAN modules. We explore three training strategies for this architecture

- **Unconstrained trainable matrix:** A standard trainable weight matrix W_{vel} maps hidden states to velocities. While flexible, this setup introduces many degrees of freedom, potentially leading to unstable or noisy velocity dynamics that could interfere with the LSTM's learning.
- **Trainable matrix with tanh constraint:** We apply a tanh activation after the weight matrix to limit velocity magnitudes. This regularization encourages more controlled velocity fields that better match plausible grid cell firing behavior.
- **Fixed random matrix:** Here, W_{vel} is initialized randomly and held constant during training. This serves two purposes: (1) it reduces noise by eliminating trainable parameters at this stage, and (2) it tests whether the LSTM can learn to produce meaningful velocity inputs through adaptation of its internal memory alone. If the matrix has a small norm, this setup also naturally bounds the resulting velocities.

We include the fixed-weight variant to investigate whether agents can develop stable spatial strategies even when the transformation from memory to velocity is static and random.

In total, we consider three variations of Architecture 1 and three of Architecture 2. Each variation may also be combined with other parameter adjustments, allowing us to systematically evaluate a wide space of designs.

4.5 Loss functions

Both the critic and the auxiliary prediction network allow for immediate verification of their estimation accuracy. The critic's output can be assessed by comparing it to the mean return over parallel trajectories, while the auxiliary prediction network's output can be directly compared to the agent's actual position. Since these are supervised learning tasks, relying solely on PPO updates for the policy is insufficient. Instead, additional loss functions must be introduced to incorporate these learning objectives.

4.5.1 Value loss

We define a standard mean squared error loss for the estimation of the value function as

$$L^{\mathrm{VF}}(\theta) = \hat{\mathbb{E}}_t \left[(V(s) - \hat{V}_\theta(s))^2 \right].$$

Here, the $\hat{V}_\theta(s)$ is the output of the neural network, while $V(s)$ is the true discounted collected reward of a rollout.

4.5.2 Entropy loss

We encourage exploration by adding a entropy loss term as well, which is common in literature and state-of-the-art models [25]. The loss function is large when the policy is more deterministic.

$$L^S(\theta) = -\sum_a \pi_\theta(a \mid s) \, \log\big(\pi_\theta(a \mid s)\big).$$

The entropy of a policy measures how spread out (or unpredictable) its action distribution is. When the policy assigns a high probability to only one or a few actions, its entropy is low. Maximizing the policy's entropy (or equivalently minimizing its negative) encourages the policy to maintain a more uniform, less "peaked" distribution over possible actions, thereby promoting exploration. By not collapsing too quickly to a single action, the policy remains stochastic, tries more varied actions, and is less likely to get stuck in a suboptimal deterministic behavior pattern.

4.5.3 Auxiliary loss

In some cases we want to have explicit path integration as well, forcing the network to improve or maintain its ability to predict its position. The auxiliary prediction is a simple MSE loss given by

$$L^{\mathrm{AUX}}(\theta) = \hat{\mathbb{E}}_t \Big[\| (\tilde{x}_t, \tilde{y}_t)(\theta) \; - \; (x_t, y_t) \|_2^2 \Big].$$

Here, $(\tilde{x}_t, \tilde{y}_t)$ is the prediction of the x and y position of the agent via the neural network, while the (x, y) is the target function with its actual position.

4.5.4 Total loss

We define $L^{\mathrm{CLIP}}(\theta) = -J^{\mathrm{CLIP}}(\theta)$ from Equation 13 as a policy loss. The total loss at each network update is then defined as

$$L(\theta) = L^{\mathrm{CLIP}}(\theta) + \alpha_1 L^{\mathrm{VF}}(\theta) - \alpha_2 L^S(\theta) + \alpha_3 L^{\mathrm{AUX}}(\theta).$$

The total loss determines how much the network is be changed at each policy-update.

4.6 Implementation

4.6.1 Algorithm

We implement Proximal Policy Optimization (PPO) generally as described in [25], introducing the following notation:

- M: the number of parallel environments (or "actors"),
- H: the number of timesteps collected from each environment during one iteration,
- K: the number of training epochs on the collected batch,
- G: the size of each minibatch,
- N: the total number of iterations.

At each iteration, we sample rollouts from all M environments, each running for H timesteps, thus creating a total of $M \times H$ transitions. We then use these transitions to compute advantages via GAE, construct minibatches, and optimize the surrogate objective over multiple epochs. After optimization, we replace θ_{old} with the newly learned parameters. Algorithm 1 provides a concise overview.

Algorithm 1 PPO (Actor-Critic Style)

for *iteration* $= 1, 2, \ldots, N$ **do**
  for *actor* $= 1, 2, \ldots, M$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for H timesteps
    Compute advantage estimates $\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_H$
  Optimize the surrogate loss L w.r.t. θ, using K epochs and minibatch size $G \leq MH$
  $\theta_{\text{old}} \leftarrow \theta$

Although the pseudocode shows the environments being sampled sequentially, our actual implementation with JAX runs the M environments in *parallel*. After computing the advantages, we collect each of the $M \times H$ transitions (along with their advantages) into a single batch. During optimization, we split this batch into randomly sampled minibatches across time steps and across different environments. For each minibatch, we compute a gradient update on the policy parameters. We repeat this minibatch sampling and update process for K epochs in total. This procedure of repeatedly splitting a dataset into random subsets is what constitutes **stochastic gradient descent (SGD)**.

Why SGD?

- **Avoiding Sharp Local Minima:** Using noisy, incomplete information from a small subset of the data can help the optimizer avoid getting trapped in sharp local minima.
- **Implicit Regularization:** The noise in the gradient-estimates can sometimes act as a natural regularizer [35], improving generalization so that the agent performs well even on unseen states.
- **Scalability:** For large datasets (or large-scale RL rollouts), it is impractical in terms of memory or compute time to perform full-batch updates.

To update the neural network weights for each minibatch, we employ the **Adam optimizer** [36], which combines momentum and adaptive learning rates to improve convergence speed and stability. Adam is particularly well-suited for reinforcement learning tasks, as it efficiently handles noisy gradients and non-stationary objectives.

4.6.2 Implementation in JAX

For simulation and training, we use the Google JAX Machine Learning Framework. JAX[1] is a small Python library that looks and feels like `NumPy`. You write the usual array formulas—addition, matrix products, nonlinear activations—and two extra pieces of magic happen automatically:

[1] https://github.com/google/jax

1. *Automatic differentiation*: given a numerical recipe for a quantity $L(\theta)$, `jax` can also provide $\nabla_\theta L$—the exact gradient we need to improve the parameters θ.
2. *Ahead-of-time GPU code*: the *first* time a function runs, `jax` records every array operation and translates the whole trace into one compact GPU program. Later calls reuse the same program, so the expensive work is done on the GPU while Python simply waits.

Both features are invisible to the user yet crucial for high-throughput reinforcement learning.

A big difference between between coding regular neural networks (such as pyTorch) and JAX, is the importance of the shape of the tensors. All arrays in the project follow a single shape convention

$$\text{shape} = (T,\ B,\ F),$$

where `T` is the number of time steps in a rollout, `B` is the number of parallel environments, and `F` is the feature dimension. Because `jax` treats the first axis as a batch, one call can crunch many time steps, many environments, or both.

However, our recurrent modules — the LSTM and the CANs—use hidden states of shape $(B, ; F)$, without the time dimension. These represent the current state and are updated recurrently. JAX slices the appropriate time step internally during recurrent processing. By wrapping the cell in `flax.linen.scan`, the LSTM is unrolled across the `T` axis ahead of time, resulting in a fused GPU kernel for the full sequence—greatly improving performance over naive looping.

One call = one PPO update. A compiled `train_step()` performs

1. interaction with the simulator for `T` steps,
2. computation of advantage estimates, and
3. several mini-batch gradient updates.

All intermediate tensors stay in GPU memory, so there is no data traffic between device and host.

The loss of actor, critic and an auxiliary task lives inside one function. Using `jax.value_and_grad` returns the scalar loss and its gradient in a single pass. Parameters are then updated with an Adam optimiser from the `optax` library. Because everything is an immutable array (a *pytree* in `jax` jargon), saving checkpoints or replicating weights across devices is trivial.

4.6.3 FAS Research Computing Clusters

All large-scale experiments were executed on Harvard's Cannon high-performance computing environment, managed by the Faculty of Arts and Sciences Research Computing (FASRC) group. Jobs are scheduled with **Slurm**. Users connect via `ssh` to a lightweight login node to submit jobs on the available partitions. In Slurm terminology a *partition* is a queue that bundles a compatible set of compute nodes together with a policy envelope (wall-time limit, memory ceilings, GPU availability, *etc.*)

Each node offers one NVIDIAH100 GPU, 16 physical CPU cores and 128GB system memory where this amount of memory is required for logging the CAN states. One node therefore offers exactly what our PPO agent needs: a single large GPU and moderate CPU throughput. The partition allows up to 72 hours of wall-clock time, enough for an agent to learn advanced tasks (in 12 and 24 hours runs the agent would not even learn "simple" tasks as to kill a cow).

Listing 1: Slurm submission script for a single-node.

```
#!/bin/bash
#SBATCH --array=0-1
#SBATCH --job-name=gru_input_hidden_layer_512_N_48_num_can_3
#SBATCH --partition=kempner_h100
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=16
```

```
#SBATCH --gpus-per-node=1
#SBATCH --time=72:00:00
#SBATCH --mem-per-cpu=8G
#SBATCH --output=./output/foraging_grid_test.out
#SBATCH --error=./output/foraging_grid_test.err
#SBATCH --mail-type=END
#SBATCH --mail-user=felix_berg@hms.harvard.edu
module load miniconda/4.12.0-fasrc01 cuda/12.3
conda activate rl-jax
export XLA_PYTHON_CLIENT_PREALLOCATE=false
srun python train.py --config configs/ppo.yaml
```

When the job reaches the top of the queue, Slurm binds the GPU and CPU cores exclusively to the process, launching the training script in an isolated `cgroup` with guaranteed resources. On this hardware the configuration from Section 4.6.2 delivers roughly 5.4×10^6 environment steps per second.

The combination of modern GPUs, generous memory, and predictable scheduling makes the `kempner_h100` partition an excellent fit for the multi-GPU `jax` experiments presented in this thesis.

4.6.4 Parameters

Table 3 lists the primary hyperparameters and constants used during training. We chose a large number of parallel environments (`num_envs`) to enhance stability in this complex task: with more environments, the policy receives more diverse and less noisy updates, reducing variance in the training process.

Table 3: Hyperparameters and constants used for training.

Parameter	Default Value	Description
`num_envs`	1024	Number of parallel environments
`total_timesteps`	3×10^9	Total timesteps
`num_env_steps`	64	Rollout length (H) per environment
`update_epochs`	4	Number of update epochs per iteration
`num_minibatches`	8	Number of minibatches per epoch
`gamma`	0.99	Discount factor
`gae_lambda`	0.8	GAE parameter
`clip_eps`	0.2	PPO clipping range
`ent_coef`	0.01	Entropy coefficient
`vf_coef`	0.5	Value function coefficient
`aux_coef`	0.1	Auxiliary (path integration) loss coefficient
`max_grad_norm`	1.0	Maximum gradient norm
`layer_size`	512	Size of hidden layers
`grid_size`	96×96	Size of the grid environment
`obs_size`	7×9	Observed state dimensions
`adam_eps`	1×10^{-5}	Adam optimizer epsilon
`adam_lr`	2×10^{-4}	Adam optimizer learning rate

Several hyperparameters, including the learning rate (`lr`), rollout length (`num_env_steps`), update epochs (`update_epochs`), and the number of minibatches (`num_minibatches`), were tuned empirically and provided stable performance in preliminary experiments. The discount factor (`gamma`), GAE parameter (`gae_lambda`), clipping parameter (`clip_eps`), entropy coefficient (`ent_coef`), and value function coefficient (`vf_coef`) follow common values found in PPO literature [25], [26]. By contrast, the auxiliary loss coefficient (`aux_coef`) is unique to this work, balancing how strongly the path integration objective influences learning.

Finally, we set the hidden-layer size (`layer_size`) large enough to allow for robust representation learning, including the creation of spatial maps. While larger sizes can improve capacity, in practice we balanced this against the computational cost to maintain reasonable training times.

5 Experiments

In this section, we evaluate and compare the training performance of different agents by analyzing their average returns. Recall that the return is defined as

$$\hat{\mathbb{E}}[\text{Return}] = \frac{1}{M} \sum_{m=0}^{M} \sum_{h=0}^{H-1} \gamma^h r_h^{(m)}.$$

where M is the number of episodes and H the episode length. In addition to return curves, we also examine the learning dynamics—specifically, what behaviors or representations the agents acquire throughout training.

5.1 Early training

We begin our analysis by examining the learning process during the first 10,000 updates, where the agent behaviors are still in their early development stages. Despite architectural differences, the general learning chronology remains consistent across all models.

Initially, the agent exhibits very short lifespans, dying quickly due to its inability to satisfy basic physiological needs or avoid predators. Even if it learns one behavior, such as eating, this does not significantly improve returns unless it also learns to drink, sleep, and evade threats. Only once these basic survival skills are in place do we observe a consistent increase in return, as the agent begins forming more advanced strategies—such as navigating sparse maps for resources or avoiding danger while foraging.

As shown in Figure 24, sleeping is typically the first reward-boosting behavior to emerge. It provides a quick reward with a single action and is therefore easily discoverable by PPO.

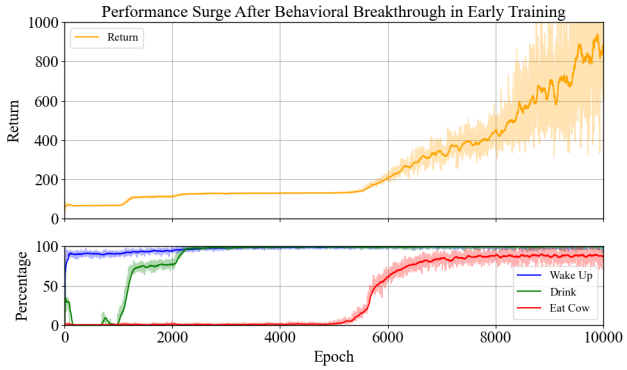


Figure 24: Early training phase: rewards begin to rise as the agent learns to sleep, drink, and finally eat.

Drinking is the second skill acquired. Unlike sleeping, drinking requires the agent to be adjacent to water and correctly execute the "DO" action (action 5). Since action 5 yields no intrinsic reward, this skill demands a correlation between perceptual input and action selection.

Eating is learned much later. It requires the agent to find a moving cow, approach it, and execute three successful "punches" (due to the cow's 3 health). This task becomes more feasible after the agent learns basic spatial tracking and coordination.

Importantly, eating and drinking often require the agent to navigate through enemy-populated areas, so acquiring these skills implies at least partial predator avoidance.

Once the agent consistently eats, drinks, and sleeps across nearly all environments, it begins learning more advanced behaviors, such as collecting resources and crafting weapons. This transition is illustrated in Figure 25, where later milestones only emerge after the foundational ones are established.

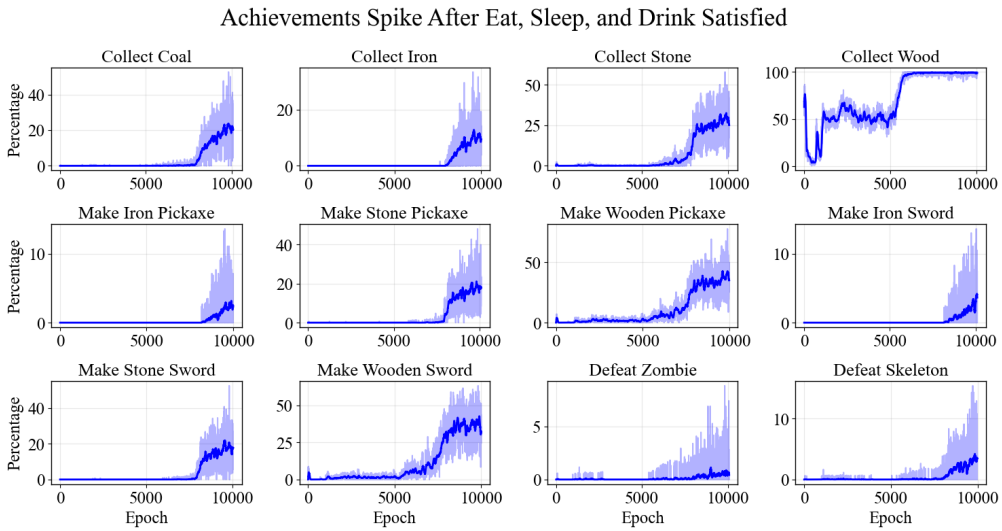


Figure 25: Progression of learned achievements over training epochs. Complex skills like "defeat zombie" or "collect stone" emerge only after mastering basic survival behaviors.

5.2 Comparing architectures

We initially compare different architectural variants of the policy network to evaluate the impact of two design choices:

- Adding auxiliary prediction heads for path integration (PI)
- Introducing sparsity in the network's fully connected layers through magnitude-based pruning.

All other hyperparameters—such as learning rate, batch size, and discount factor—are kept constant across runs. Each agent is trained for over 80,000 epochs, with each epoch processing rollouts from all parallel environments.

.

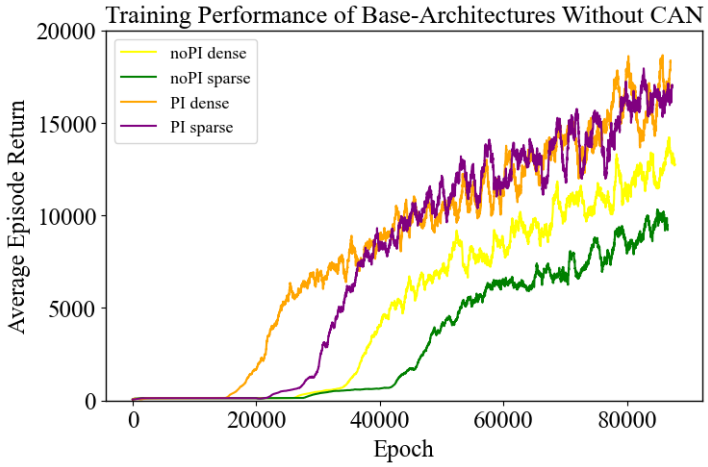


Figure 26: Training curves for four different architectures over more than 80,000 epochs.

Figure 26 presents the average episode return throughout training. The x-axis tracks the number of epochs, while the y-axis reports the mean return per epoch across all environments.

The results show that agents equipped with auxiliary path integration (PI) heads outperform their non-PI counterparts. One possible explanation is that encouraging the agent to maintain a sense of position helps

it associate resources with specific locations, enabling more efficient navigation and planning.

Introducing sparsity slightly impairs the performance of the non-PI agent, while the PI agent remains largely unaffected. This suggests that pruning up to 90% of the weights can preserve competitive performance, offering a promising approach to reducing memory and computational costs without significantly compromising learning efficacy.

Architectures with Grid Module:

We now exclude dense networks from consideration, as earlier results showed that sparse networks can achieve comparable performance. Thus, Baseline PI refers to the sparse path-integrating agent, and Baseline noPI refers to the sparse non-path-integrating agent.

Figures Figure 27 and Figure 28 show that adding CAN modules has little effect on overall performance. Among the variants, the LSTM-CAN PI noTrain (orange line) appears to slightly outperform the Baseline PI. Notably, this architecture has the fewest trainable parameters among the LSTM-input variants, which could contribute to its stable performance.

In contrast, the LSTM-CAN PI noTanh (red line), which allows unconstrained velocity outputs and has many trainable parameters, performs slightly worse than the baseline. This suggests that unregulated velocity magnitudes may disrupt the grid module's contribution.

A deeper analysis of how these velocities evolve and what information the CANs encode will be presented in Part 2 of the thesis, where we explore the memory dynamics and spatial representations of these architectures.

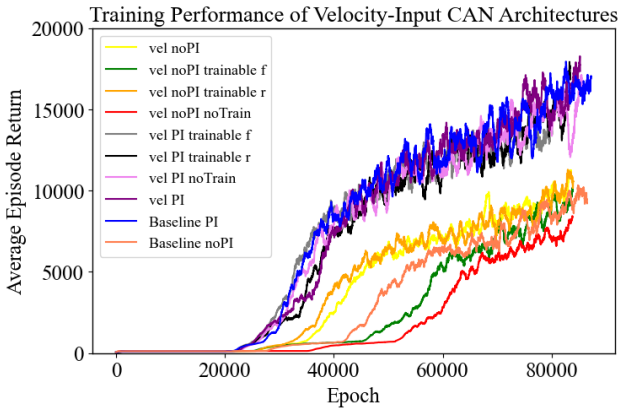


Figure 27: Performance of variations of architecture 1.

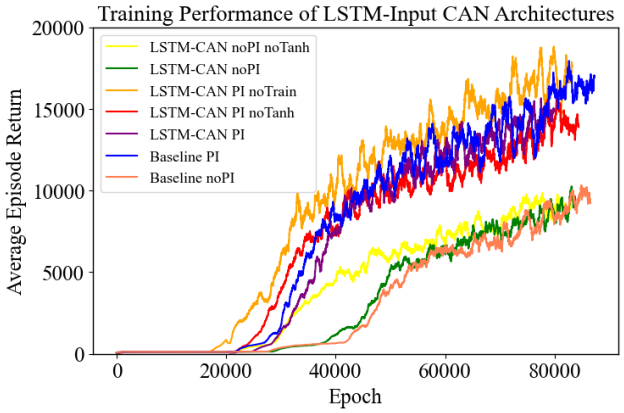


Figure 28: Performance of variations of architecture 2.

Traceplot and Agent Movement:

In Figure 29, we show a trace plot from late-stage training of the sparse path-integrating agent. Events such as eating, drinking, and predator sightings are highlighted along the trajectory. We observe that the agent explores a large portion of the map, rather than remaining near a single patch with nearby water. This indicates adaptive behavior: once local resources are depleted, the agent seeks out new patches and ponds elsewhere. The red markers—indicating predator encounters—tend to form short lines, suggesting the agent successfully escapes threats rather than getting trapped or killed.

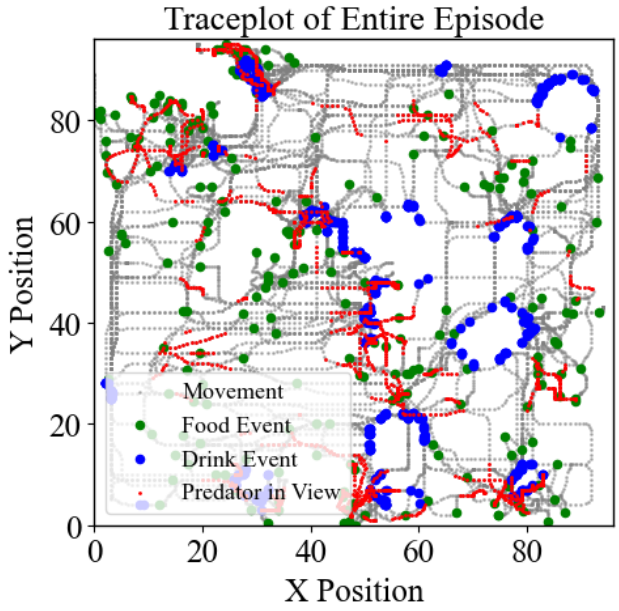


Figure 29: Trace plot of the sparse path-integrating agent. Colored dots show food, water, and predator encounters. Grey overlay is a five-step moving average to indicate trajectory smoothness.

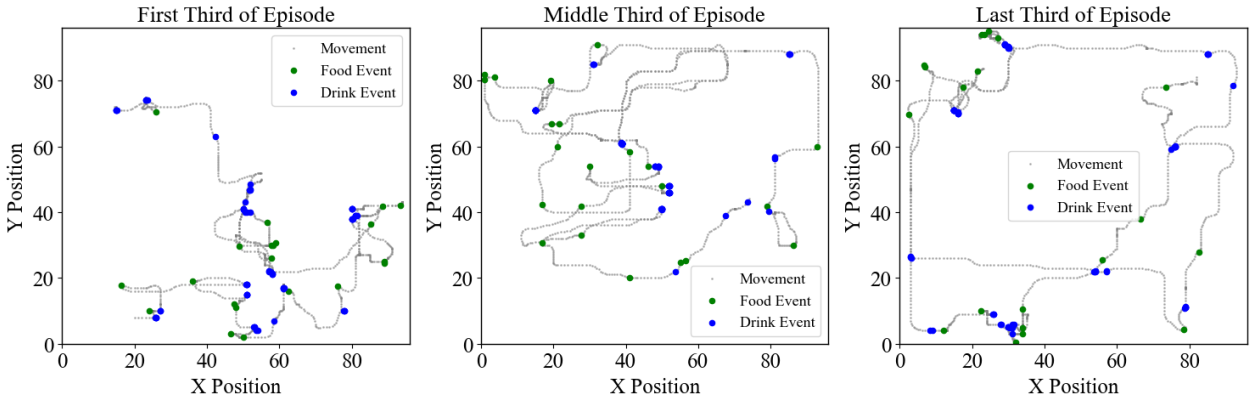


Figure 30: Agent trajectories during the first, middle, and last thirds of an episode. Movement becomes more directed and strategic over time.

To better understand the agent's evolving strategy during an episode, Figure 30 shows movement patterns split into the first, middle, and final thirds of a single trajectory. In the early phase, the agent appears to be in exploratory mode—visiting various regions, drinking frequently, and opportunistically eating when it finds cows. By the middle third, its paths become more structured, covering broader areas with straighter movement. In the final third, its behavior becomes even more goal-directed. The agent appears to revisit known patches, suggesting that it has mapped out the environment and is actively exploiting its internal representation to maintain survival.

These visualizations highlight that the agent not only discovers and remembers resource locations, but also adapts its behavior based on past experience. It avoids predators efficiently, searches for new resources when local ones are depleted, and exhibits clear transitions from exploration to exploitation. Overall, even under the constraint of sparse path integration, the agent learns to navigate large environments using structured and effective policies.

Part II

Statistical Analysis of Space Representation

6 Theory — Memory Analysis

To investigate memory and spatial representation, we log detailed traces during simulation and training while keeping the network weights θ fixed. Specifically, we record:

- **Environmental Data:** Information about the agent's surroundings, including positions, actions, predator presence, and other map features.
- **Neural Activity Data:** The internal states of the agent's neural network, including hidden states from the LSTM and, if present, the CAN modules.

Using the environmental data, we analyze behavior by defining and labeling *patches*—map locations where the agent has consumed resources multiple times. We then quantify how often and under what conditions the agent revisits these locations. This allows us to study spatial preferences and decision-making patterns using behavioral generalized linear models (GLMs).

From the neural activity data, we examine whether the LSTM encodes spatial memory by decoding positional information from its hidden states. We assess both population-level encoding and individual neuron selectivity, identifying neurons that contribute significantly to spatial representation.

Additionally, we explore the role of the continuous attractor networks (CANs). Although earlier results showed that CANs had minimal impact on training performance, we now assess their internal dynamics and contributions. We analyze correlations, receptive fields, and how CAN states evolve over time to determine whether and how the policy makes use of them.

The following sections provide the theoretical background for the behavioral and neural decoding analyses. Much of the theory in Sections 6.1, 6.2, and 6.3 was originally introduced in [19], though modifications have been made for this thesis.

6.1 General Linear Models

A Generalized Linear Model (GLM) is an extension of ordinary linear regression that allows the dependent variable (often called the response) to have an error distribution other than the normal distribution. GLMs are widely used when your data come from different distributions in the exponential family (e.g., Normal, Binomial, Poisson, Gamma).

Mathematically, a GLM can be expressed as:

$$Y \sim \text{Distribution in exponential family}, \quad g(\mu) = \eta = X\beta, \quad \mu = E[Y].$$

Here Y is the response variable, X is the matrix of predictors, β is the vector of coefficients and $g(\cdot)$ is the link function which connects the linear predictor to the mean response μ.

The goal of the agent is to survive and in order to survive it needs to find food, which is found within a patch. We want to analyse how the agent chooses to visit patches that it has already seen before. We therefore define the variable $Y \in \{0, 1\}$ as the patch-rivisitation variable that is zero if the patch is not revisited and is one if the patch is revisited.

The response is binary so it is natural to use the binomial model.The link function is then

$$\log\left(\frac{\mu}{1-\mu}\right) = X\beta,$$

where $\mu = P(Y = 1|X)$.

6.2 Decoding Spatial Representations from Hidden States

To investigate the agent's memory and its ability to predict spatial positions, we analyze whether the hidden states encode information about past, present, and future locations. If an agent can maintain an internal representation of position, it may leverage this memory to generate a movement plan.

During the late stages of training, we recorded the agent's hidden states h_t for entire episodes, alongside its spatial position (x, y). Our goal is to decode the agent's position $Y_{t+\Delta t}$ at a future or past timestep Δt using only the hidden states at time t. Formally, we aim to learn a mapping function f from the hidden state space to position:

$$Y_{t+\Delta t} = \begin{bmatrix} \Delta x_{t+\Delta t} \\ \Delta y_{t+\Delta t} \end{bmatrix} = f(\vec{h}_t), \quad h_t \in \mathbb{R}^{512}.$$

This function allows us to assess whether the agent's internal state contains information about spatial navigation, potentially similar to biological representations.

Regularization and Model Selection
Since certain neurons remain inactive for entire episodes, the coefficients of f can become large, leading to an ill-conditioned problem. To mitigate this, we introduce a regularization term that penalizes large coefficients, resulting in the following loss function:

$$L_{\Delta t}(f) = \sum_{i=1}^{N} (Y^i_{t+\Delta t} - f(h^i_t))^2 + \alpha \|f\|^2_K.$$

This formulation ensures that the model remains well-posed while preserving interpretability.

For each Δt, we need to fit a separate function f. The choice of model is guided by three key criteria:

1. **Interperatibility**: The ability to analyze how individual hidden states contribute to position encoding.
2. **Performance**: The accuracy of position predictions.
3. **Computational Efficiency**: The feasibility of training the model on large datasets.

Comparing Different Models
Selecting f as a neural network would likely yield high predictive performance but would obscure the contributions of individual neurons and require longer training times. Kernel methods, such as the Radial Basis Function (RBF) kernel, offer good interpretability and strong performance but suffer from high computational costs:

$$f(x) = \sum_{i=1}^{N} \alpha_i K(x, x_i), \quad K(x, x_i) = \exp(-\gamma \|x - x_i\|^2).$$

The generation of f requires inverting an $N \times N$ matrix, leading to a time complexity of $O(N^3)$, which is infeasible for tens of thousands of data points per Δt.

To balance interpretability, efficiency, and accuracy, we use Ridge Regression:

$$f(h_t) = Ah_t + b, \quad A \in \mathbb{R}^{2\times 512}, \quad b \in \mathbb{R}^2.$$

This approach has a time complexity of $O(Np^2)$, where p is the feature dimension, making it significantly more scalable. Additionally, linear regression provides a clear mapping between each hidden state and spatial position, allowing for single-neuron analysis. Although its performance does not match that of kernel methods, it remains sufficient for qualitative analysis.

For future analysis, incorporating kernel-based methods could offer valuable insights. These approaches can capture complex, nonlinear relationships between neural activity and spatial position—potentially revealing differences in spatial representations across architectures that linear methods may miss. While linear regression serves well for interpretability and neuron-level attribution, kernel methods may uncover subtler representational structures critical for understanding architectural distinctions.

6.3 Single Neuron Decoding Theory

The contribution of a single neuron $h_t^{(i)}$ to spatial representation may be related to specific patterns of activity. For example, certain clusters of neurons may exhibit increased activation in response to environmental factors, such as the presence of predators. In this section, we analyze the distribution of individual neuronal contributions to the agent's position prediction.

Given our linear predictor:

$$Y_{t+\Delta t} = Ah_t + b,$$

the predicted x-coordinate is expressed as:

$$x_{t+\Delta t} = \sum_{i=1}^{512} A_{1i} h_t^{(i)} + b_1.$$

Each neuron's contribution to the prediction depends on both its activation dynamics $h_t^{(i)}$ and the corresponding coefficient in the fitted model A_{1i}. To illustrate this, we examine the contributions of the first five neurons from a randomly selected episode for $\Delta t = 10$.

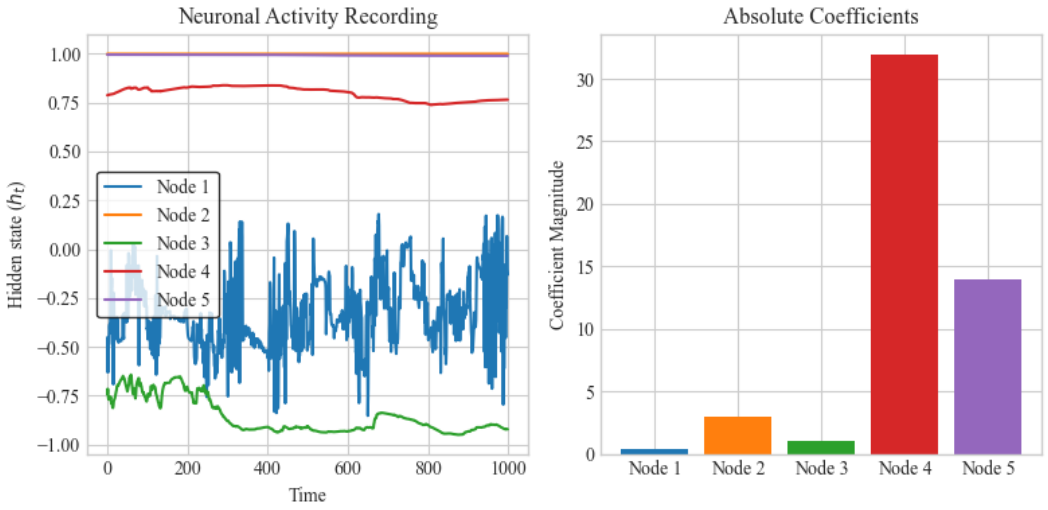


Figure 31: Left: Hidden state activity over time. Right: Magnitude of the fitted coefficient A_{ij} in the predictive model. The plot indicates that neurons with high activation variance tend to have smaller fitted coefficients.

From Figure 31, we observe that both neuron 1 and neuron 4 contribute significantly to variations in the predicted x-coordinate. Neuron 1 exhibits high variability in its hidden state h_t, while neuron 4 has a larger corresponding coefficient A_{1i}. The total contribution of a neuron to the prediction is determined by the product of these two factors.

To quantify the contribution of individual neurons, we define $\sigma_h^{(i)}$ as the standard deviation of $h_t^{(i)}$ within an episode, and let σ_t denote the vector of standard deviations across all hidden states.

In our linear model, the variation in prediction attributable to neuron i can be expressed as:

$$\text{Contribution}_i = \begin{bmatrix} |A_{1,i}| \\ |A_{2,i}| \end{bmatrix} \sigma_t^{(i)}.$$

For example, in the prediction of $x_{t+\Delta t}$, a large value of $A_{1,i}\sigma_t^{(i)}$ indicates that neuron i has a strong influence in the prediction of $x_{t+\Delta t}$. We define this term as the contribution of neuron i to the prediction.

6.4 Decoding Grid Cells

After training neural networks that include Continuous Attractor Network (CAN) modules, we focus on three key areas of analysis:

i) How the velocity vector $\vec{v}$ is generated,

ii) What information is encoded by the bump position in the CAN, and

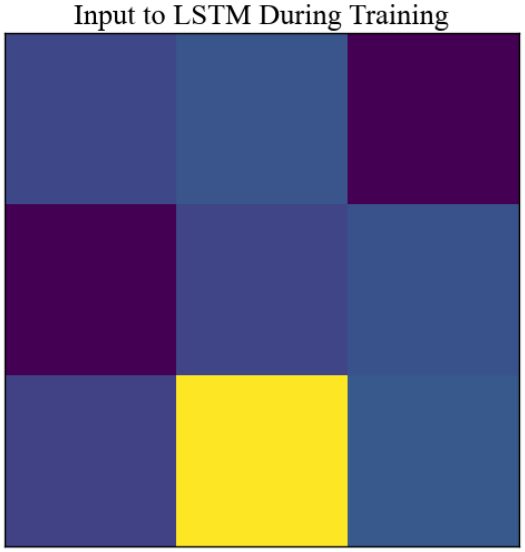
iii) How the LSTM neurons utilize the CAN activity.

The third point—how LSTM neurons respond to CAN outputs—can be interpreted through the lens of receptive fields. That is, we aim to characterize the specific patterns in CAN activity that drive LSTM neuron activations.

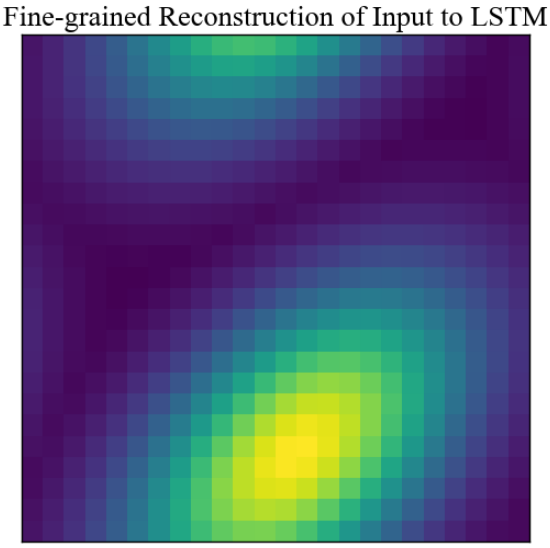
To investigate this, we reconstruct the input from the CAN to the LSTM at a finer spatial resolution than was used during training. Our hypothesis is that the dimensionality reduction applied to the CAN before feeding it into the LSTM does not lead to any information loss. That is, the lower-dimensional input used during training (shown in Figure 32(a)) preserves the same spatial information as the full-resolution version (shown in Figure 32(b)).

Although the training input (Figure 32(a)) does not visually resemble a bump-like activation, we reconstruct it into the higher-resolution bump representation in Figure 32(b) for subsequent correlation analysis. If our hypothesis holds, then correlating the LSTM hidden states with the reconstructed, fine-grained CAN activity should yield smooth spatial correlation maps—ideally exhibiting bump-shaped profiles.

Figure 32 illustrates these two representations from actual training. They depict the same underlying function, sampled at different resolutions. Panel (a) is the low-dimensional input used during training, while panel (b) shows the full-resolution reconstruction. Since we assume that both representations contain equivalent information, we use the latter to perform a more detailed receptive field analysis of the LSTM units.



(a) Training input (reduced CAN)



(b) High-resolution reconstruction

Figure 32: (a) shows the low-dimensional CAN output fed into the LSTM during training. (b) shows a higher-resolution reconstruction of the same CAN activity used for correlation analysis.

6.4.1 Architecture 1

i) Velocities. In Architecture 1, the agent's velocities are directly determined by its previous actions. We refer to the transformation from actions to velocities as the application of *velocity scales* or *frequencies*. For instance, a frequency of 2 implies that a translation action results in a velocity vector with magnitude $|\vec{v}| = 2$. We use the term "frequency" because higher velocities cause the activity bump in the continuous attractor network to traverse the rhombus-shaped sheet more rapidly, leading to firing patterns with shorter spatial periods.

As introduced in subsection 4.4, we train three distinct strategies to generate velocity scales in this architecture. In the analysis in section 7, we examine how each strategy translates actions into velocities and how these velocities are distributed.

ii) Firing patterns. Since the bump dynamics in Architecture 1 are driven solely by velocities derived from movement-action, it naturally encodes the agent's spatial position over time. To demonstrate this, we will visualize the firing patterns of individual grid cells resulting from the velocity scales described in part (i). Specifically, we allow the agent to traverse the map and record the activity of one representative neuron in each continuous attractor network (CAN). For each tile in the environment, we compute the neuron's average activation and present the results as a heatmap. This provides a clear spatial visualization of grid-cell activity in response to the agent's motion.

iii) LSTM receptive fields. Finally, we analyze how spatial information from the CANs is processed by the downstream LSTM. We do this by measuring the Pearson correlation between each CAN unit and the LSTM's hidden state. Letting h_t denote the LSTM hidden state and y_t the CAN activity at time t, the Pearson correlation coefficient is defined as:

$$r_{HY} = \frac{\sum_{t=1}^{n}(h_t - \bar{h})(y_t - \bar{y})}{\sqrt{\sum_{t=1}^{n}(h_t - \bar{h})^2}\sqrt{\sum_{t=1}^{n}(y_t - \bar{y})^2}}$$

To assess statistical significance, we compute a two-tailed p-value under the null hypothesis $H_0 : r = 0$. Given the sample correlation r and sample size n, the test statistic is:

$$t = r \cdot \sqrt{\frac{n-2}{1-r^2}} \tag{18}$$

This statistic follows a Student's t-distribution with $n - 2$ degrees of freedom. The p-value is calculated as:

$$p = 2 \cdot P(T \geq |t|), \quad T \sim t_{n-2} \tag{19}$$

This p-value indicates how likely it would be to observe a correlation as strong as r purely by chance, assuming there is no true relationship between the variables.

The smaller the p-value, the stronger the evidence against the null hypothesis:

- $p < 0.05$ suggests that the correlation is statistically significant at the 5
- $p \geq 0.05$ implies that the correlation may be due to random chance, and we fail to reject the null hypothesis.

For our correlation tests, we use a sample size of $n = 8002$. The only unknown parameter in equations 18 and 19 is therefore the correlation coefficient r. Solving these equations for r yields that $|r| > 0.022$ corresponds to a statistically significant correlation. We note that this is a very low threshold, which means we are sensitive to even weak linear relationships.

6.4.2 Architecture 2

i) Velocities. In architecture 2, the velocities are generated through a learned weight matrix W_{vel}, meaning that what the velocities represent is entirely determined by training. We will analyze how these velocities are structured across the three strategies introduced in subsection 4.4. To assess whether the learned velocities behave in a meaningful way, we inspect their mean and standard deviation. This provides insight into whether the strategy produces sensible movement patterns and whether the approach might be promising for future exploration.

ii) Firing patterns. Unlike architecture 1, the CANs in architecture 2 are not forced to encode spatial position—they are free to represent any information that supports behavior. To investigate this, we compute the Pearson correlation between the CAN activations and a set of behavioral variables. This allows us to identify whether the CANs capture meaningful patterns or latent behavioral properties.

iii) LSTM receptive fields. Finally, we evaluate how much the LSTM depends on the information provided by the CANs. To do this, we compute the Pearson correlation between each CAN unit and the LSTM hidden states. This analysis indicates to what extent the network is integrating CAN output into its memory and decision-making processes.

7 Results — Memory Analysis

In this section, we present the results of the memory analysis conducted on the trained agents. We begin by examining how the agent's internal state and environmental features influence patch revisitation behavior, providing insight into whether the memory architecture supports strategic foraging.

Following this, we evaluate the encoding of spatial information using both single-neuron and multi-neuron decoding analyses. These analyses allow us to assess how well spatial position is represented across different architectural configurations. The primary comparisons are between sparse versus non-sparse networks, and agents with versus without explicit path integration.

We note that the inclusion of continuous attractor networks (CANs) did not significantly impact the quality of spatial representations or decoding performance at the single-neuron level. As such, most of the focus will remain on the effects of sparsity and path integration. Nevertheless, we conclude the section with an exploratory analysis of the CANs to investigate their internal dynamics and potential roles beyond spatial encoding.

Portions of the early subsections build on material previously presented in [19], though the large majority of the results shown here are novel.

7.1 Behavioural Decision GLM

We recall from subsection 6.1 that we use the following model for behavioral analysis:

$$\log\left(\frac{\mu}{1-\mu}\right) = X\beta, \quad \mu = P(Y = 1|X).$$

To understand the factors influencing patch revisitation, we collect various features such as hunger level, distance to the patch, and other environmental variables to construct the predictor matrix X. The binary outcome variable Y indicates whether a patch was revisited, and we fit a binomial Generalized Linear Model (GLM) to estimate the coefficients in β. These coefficients, along with their 95% confidence intervals, are visualized in Figure 33.

From the plot, we observe that higher food consumption (eatRate) reduces the likelihood of revisiting a patch, likely due to resource depletion. However, water does not deplete, so increased drinking behavior is associated with a higher probability of returning to a patch.

Interestingly, seeing predators and experiencing hunger do not significantly affect patch revisitation behavior. Instead, the agent shows a preference for patches that are spatially and temporally close. Spatial proximity

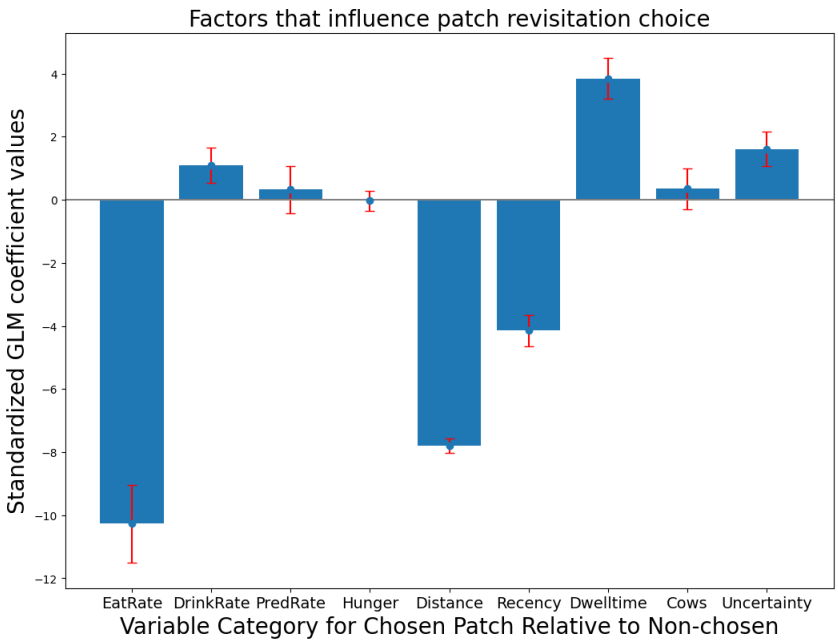


Figure 33: Bar plot with 95% confidence intervals showing standardized GLM coefficient values for factors influencing patch revisitation. A more positive coefficient indicates a greater likelihood of revisiting a patch.

is expected, as closer patches require less effort to reach. Temporal recency also plays a role, which aligns with our later findings on memory decay—the agent's spatial memory is strongest for recent locations.

Additionally, the amount of time spent in a patch increases the probability of returning, suggesting that longer visits reinforce the importance of the location.

One of the most intriguing factors is uncertainty, which reflects how well the agent performs in the auxiliary position-prediction task. The model suggests that when the agent struggles to estimate its position accurately (i.e., when auxiliary loss is high), it is more likely to return to the patch. A possible explanation is that the agent actively seeks locations where its path integration updates are less reliable, potentially allowing it to refine its internal spatial model. This behavior suggests that the agent may be leveraging re-visitation as a way to improve localization and memory stability.

7.2 Neural decoding — Correction for Model Biases

Recall from the theoretical discussion that our goal is to determine whether the agent's hidden states $h_t \in \mathbb{R}^{512}$ encode spatial information. Specifically, we seek a linear transformation (A) and bias term (b) that predict the agent's position $Y_{t+\Delta t}$ at a future (or past) timestep Δt:

$$Y_{t+\Delta t} = A\, h_t + b.$$

A low prediction RMSE indicates that the hidden states effectively capture position-related information. We explore various values of Δt to assess short-term memory (small $|\Delta t|$) and longer-term memory (large $|\Delta t|$).

In the following subsection, we identify key violations of model assumptions stemming from the continuous and auto-correlated nature of the hidden-state data—issues that must be addressed to obtain reliable decoding results. The subsequent section then presents the corrected decoding outcomes, comparing architectures and examining how spatial representations evolve throughout training.

7.2.1 Problems with Normal Ridge Regression on an Episode

Decoder performance quantifies how well the agent's hidden states (h-states) encode its position on the map. A decoding error of 0 indicates perfect knowledge—i.e., the position is a linear function of the hidden state

with no residual error. Conversely, an error equal to the agent's average displacement corresponds to complete spatial ignorance, where predictions are no better than random guessing. If the error exceeds the average displacement, it may suggest overfitting to the training set. For a well-performing agent, we therefore expect decoding performance to fall between 0 and the average displacement, with the value indicating how many tiles the decoder typically misplaces the agent. We also except that when the agent tries to predict far into the past or future, the resulting RMSE is no better than the average displacement.

We begin with a standard decoding setup: a random train/test split, using 75% of the hidden states for training and the remaining 25% for testing. We evaluate the decoder on a single episode. The results are shown in Figure 34.

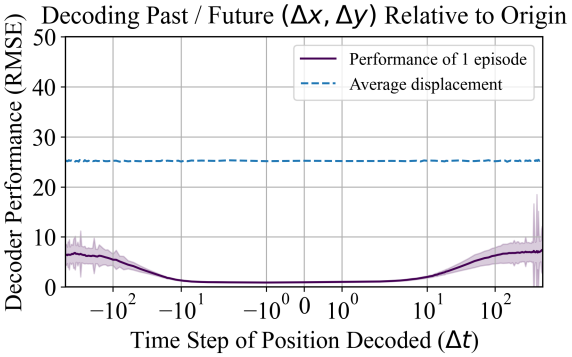


Figure 34: RMSE of one episode with a random train-test split. We observe that the RMSE is too low for $|\Delta t| \approx 1000$ to be plausible, indicating that model assumptions for decoding are violated.

What we see is that the agent has almost perfect knowledge of its current position and its position $\Delta t \in (-10, 10)$. We also see that it has very good knowledge of position 1000 timesteps into the future and into the past. However, it seems very unreasonable for a model to be able to predict position when $|\Delta t| \to \infty$, and we should assume that for large $|\Delta t|$, the performance should be the same as the average displacement or worse. The reason for this is that a condition for the ridge regression is that the data are i.i.d. However, we can analyse the properties of the feature space and target space to see why this is not true.

7.2.2 Problems with Continuous, Temporal Auto-Correlated Data

A central challenge in decoding spatial position from hidden states is that both the features (h_t) and the target positions evolve continuously over time. The hidden states are high-dimensional and strongly temporally correlated—adjacent states are very similar, and similar states tend to occur only within a narrow temporal window. This smoothness, combined with a random train-test split, allows the model to exploit temporal structure and implicitly encode time.

Problem with Features: Continuity and Uniqueness Figure 35 shows the L_1 distance between hidden states across 2,000 consecutive timesteps. The following observations can be made:

- Consecutive hidden states (e.g., h_t and h_{t-1}) are highly similar, confirming that the feature space is smooth and continuous.
- Hidden states at distant timepoints (e.g., h_{2000}) are relatively distinct from earlier states, indicating temporal uniqueness.
- This structure allows a decoder to approximate the temporal index—or "timestamp"—of each hidden state, even within a single episode, enabling unintended time decoding.

Problem with Outputs: Continuity and Interpolation As shown in Figure 36, both the predicted and actual position trajectories are continuous in space and time. This allows the model to interpolate smoothly between training examples. Rather than generalizing based on meaningful spatial structure, the decoder may instead exploit continuity in the data, producing interpolated outputs that fall close to—but not necessarily aligned with—actual agent positions.

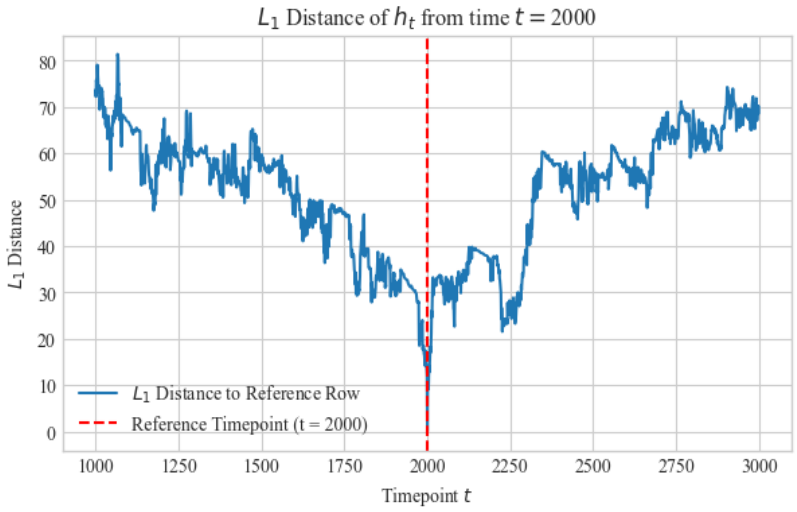


Figure 35: L_1-based similarity search among hidden states over 2,000 timesteps. Nearby timepoints yield nearby vectors in feature space.

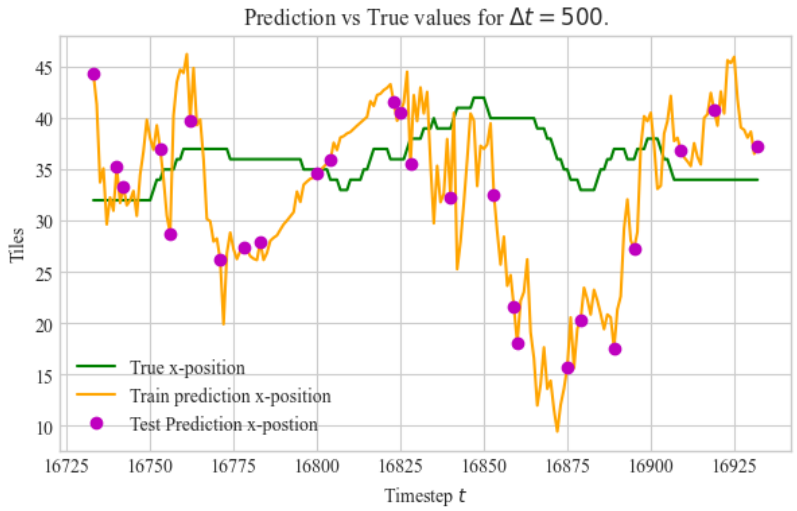


Figure 36: Predicted test trajectories versus training data for a single episode. The model tends to interpolate between temporally adjacent points.

7.2.3 Addressing the Feature Space Problem: Training on Multiple Episodes

Training on a *single* episode allows the model to implicitly encode time in the hidden states h_t, since each timepoint is unique and temporally correlated. To mitigate this issue, we train the decoder on *multiple* episodes. This introduces greater variability in the hidden state trajectories, reducing the decoder's ability to rely on time-based cues alone.

Figure 37 shows how the root-mean-square error (RMSE) changes as more episodes are included in the training set.

We observe that increasing the number of episodes leads to a general rise in RMSE, indicating that the model can no longer exploit implicit time encoding as effectively. Nonetheless, even with 30 training episodes, the RMSE for intermediate Δt values remains significantly below the average displacement baseline. This suggests that some meaningful positional information is still preserved in the hidden states.

Predictions at large temporal lags ($\Delta t \to 1000$) still perform better than expected by chance, indicating that

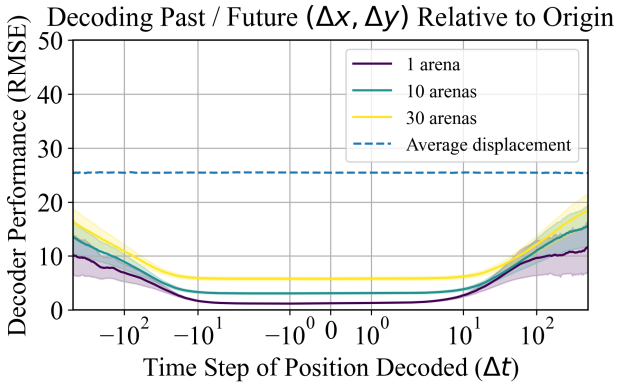


Figure 37: RMSE across increasing numbers of training episodes. As more episodes are added, it becomes harder for the model to decode position purely from implicit time signals, resulting in higher overall RMSE.

temporal autocorrelation and data leakage issues remain only partially resolved.

7.2.4 Addressing the Output Space Problem: Temporal–Chronological Split in Train and Test Data

To ensure that decoding performance reflects genuine generalization rather than interpolation, we restructure the train–test split to enforce temporal separation. Instead of randomly shuffling timepoints, we split each episode chronologically: the first 75% of timesteps are used for training, and the remaining 25% are reserved for testing. This strategy prevents the decoder from leveraging temporally adjacent samples and better evaluates its ability to *extrapolate* to truly unseen data.

This approach also complements the solution to the feature space problem by being applied across multiple episodes, thereby reducing overfitting to specific temporal patterns.

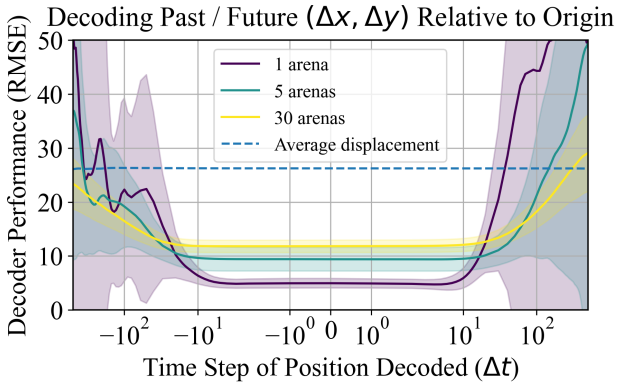


Figure 38: RMSE of a sparse path-integrating agent, evaluated using a temporally structured train–test split. This setup corrects for unrealistic model assumptions in continuous data.

As shown in Figure 38, the RMSE now converges toward the agent's average displacement for large $|\Delta t|$, as expected. This suggests that the revised evaluation setup yields a more faithful assessment of positional encoding in the hidden states. The RMSE levels off at approximately 10 tiles, indicating moderate spatial accuracy.

Interestingly, for time shifts within the interval $\Delta t \in (-100, 100)$, the RMSE remains below the average displacement. This may reflect the agent's ability to retain short-term spatial memory or anticipate near-future locations—suggesting a meaningful encoding of position within this local temporal window.

In summary, obtaining a realistic measure of spatial encoding requires training on multiple episodes and

using a temporally ordered train–test split. These steps are essential to prevent the decoder from exploiting time correlations and to accurately assess how well the agent represents its position.

7.3 Neural Decoding — Spatial Representation Results

Having corrected for methodological biases in the previous section, we now turn to the core decoding results to assess how spatial information is represented in the agent's hidden states. Using the revised training protocol—multiple episodes and temporally ordered splits—we evaluate how well different architectures encode both absolute and relative position.

This section begins by analyzing the ability of agents to represent relative spatial information, followed by a temporal analysis of spatial encoding throughout training. We then compare decoding performance across architectures to assess the effects of path integration and sparsity. Finally, we examine whether adding continuous attractor networks (CANs) improves positional awareness, and conclude with a summary of the key findings.

7.3.1 Comparing RMSE Relative to Origin

We now compare the decoding performance of different base architectures with respect to path integration and sparsity. The results reveal clear distinctions in how well each model encodes positional information.

As shown in Figure 39, the non-path-integrating (nPI) agent fails to learn an accurate representation of position. Without an explicit training signal for location, it does not implicitly develop a robust spatial encoding. The dense path-integrating (PI) agent performs somewhat better, suggesting that the inclusion of a path-integration loss helps guide the learning of position-related features. However, the sparse PI agent outperforms both alternatives, likely due to its architectural capacity to assign specific neurons to encode spatial structure more efficiently.

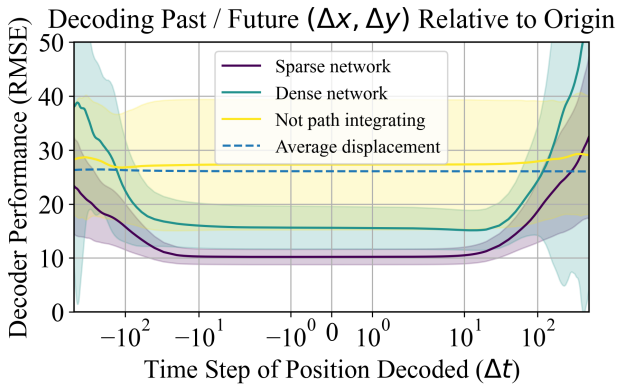


Figure 39: RMSE comparison between sparse PI, dense PI, and non-PI agents.

Both path-integrating agents demonstrate strong positional awareness for short-term predictions within the range $|\Delta t| < 50$. This suggests that the agents maintain a short-term memory of past movement and are capable of forming near-future positional plans based on recent experience.

7.3.2 Relative RMSE

Since the sparse path-integrating agent exhibited the highest level of positional awareness among the tested architectures, we focus this analysis on that model to investigate whether it also encodes *relative* position. In this context, relative position refers to the displacement vector from the agent's current location to its position Δt timesteps in the future. This shifts the decoding task from predicting absolute coordinates to capturing the direction and magnitude of movement over time.

Figure 40 displays the root-mean-square error (RMSE) for these relative position predictions, evaluated across a range of time lags. The dotted line represents the agent's average relative displacement, serving as a baseline that would be expected if the agent had no informative internal representation.

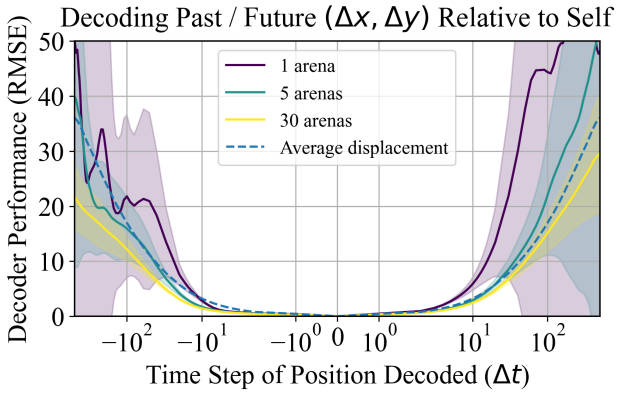


Figure 40: Relative-position RMSE over varying Δt for the sparse path-integrating agent. The dotted line indicates the agent's average relative displacement. Prediction error largely follows this baseline, suggesting weak relative encoding.

For most values of Δt, the RMSE remains within the confidence interval of the average displacement, suggesting that the agent's hidden states carry only weak signals about relative position. This indicates that the model is not explicitly encoding the direction or magnitude of future movement in a way that supports reliable decoding.

However, an exception appears in the short-term past ($\Delta t \in (-20, 0)$), where the RMSE dips slightly below the average displacement. This suggests that the agent retains some memory of recent movements, allowing it to better estimate where it has just come from. Still, this effect is transient, pointing to the fact that detailed relative positioning is not a central component of the agent's internal model. Instead, it appears that absolute location—rather than relative displacement—is more strongly represented, likely because it plays a more critical role in task performance and reward acquisition.

7.3.3 Spatial Representations Across Training

To understand how spatial representations evolve during learning, we track the positional encoding capabilities of the sparse path-integrating agent across different stages of training. Given that this agent previously showed the strongest spatial awareness, it provides a useful case study for examining how such representations emerge and improve over time.

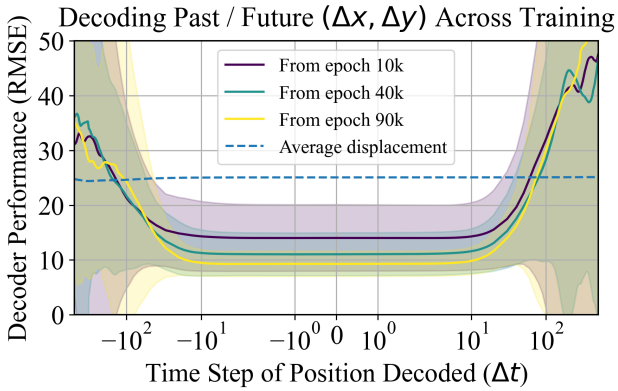


Figure 41: RMSE for absolute position decoding at various stages of training, showing improved spatial representations over time.

We apply the same linear decoding procedure as before, evaluating how well the hidden states predict absolute position. However, care must be taken to ensure fair comparisons across training epochs: later in training,

episodes tend to be longer, meaning that simply decoding a fixed number of episodes would introduce bias due to unequal data availability.

To control for this, we extract the first 20 episodes from epoch 10,000 that each have a length greater than 6,000 timesteps. From these, we use a consistent window of timesteps 500 to 5,500 for the decoding analysis. This ensures that all decoding models are trained on comparable datasets across epochs. The resulting RMSE values are shown in Figure 41.

The results reveal a clear trend: as training progresses and the agent's overall performance improves, its ability to encode spatial position becomes more accurate. This suggests that the agent incrementally refines its internal spatial model in conjunction with learning to solve the task.

7.3.4 RMSE with CANs

We also examine whether adding continuous attractor networks (CANs) enhances spatial encoding in the agent's hidden states. As shown in Figure 42, we compare the sparse path-integrating (PI) agent with three different CAN-based strategies.

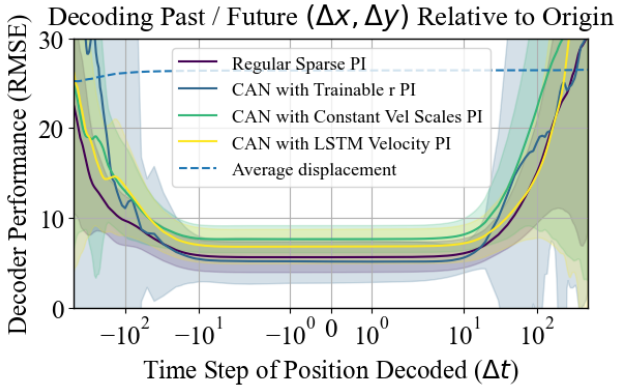


Figure 42: Decoding RMSE for the standard sparse PI agent versus three CAN-enhanced variants.

Across all tested architectures, the addition of CANs does not lead to a measurable improvement in position decoding. The RMSE curves for the CAN-enhanced models largely overlap with that of the baseline, and the confidence intervals are broad, limiting the strength of any conclusions.

However, one notable trend is that the confidence bounds for the CAN-augmented agents are consistently wider than those of the baseline. This may suggest that the CAN introduces additional variability or noise into the memory representations, which could interfere with the network's ability to form stable spatial encodings.

7.3.5 Summary of Encoding Findings

Our analysis reveals several important insights into how spatial information is represented in the agent's hidden states:

- **Path integration enhances short-term spatial encoding**: Agents trained with an explicit path integration objective (PI agents) show markedly better performance in predicting positions within a short temporal window. This indicates that path integration facilitates the development of internal spatial representations.
- **Hidden states are continuous and temporally auto-correlated**: The smooth, high-dimensional structure of the hidden states enables the model to infer time from hidden state trajectories, especially when decoding is performed on a single episode. Training on multiple episodes mitigates this issue, forcing the model to rely more on genuine spatial information than implicit time encoding.
- **Chronological splitting reveals true positional knowledge**: Using a temporally structured train–test split removes the decoder's ability to interpolate between temporally adjacent states. Under

this setting, PI agents still perform well for $\Delta t \in (-100, 100)$, demonstrating a meaningful, short-range encoding of position.

- **Relative position is weakly represented**: Although the agent shows some awareness of recent past movement ($\Delta t \in (-20, 0)$), its hidden states primarily encode absolute position. This may be due to the consistent grid tile layout across environments, making absolute positioning more directly beneficial for navigation—akin to how some animals use global cues like geomagnetic fields [37].
- **CANs do not improve linear spatial decoding**: Incorporating continuous attractor networks did not enhance position decoding, and in some cases increased variance. This suggests that CANs may introduce noise or encode spatial features in a nonlinear manner that is not captured by linear regression.
- **Spatial representations improve with training**: For the sparse PI agent, decoding accuracy improves over the course of training. This indicates that spatial awareness develops alongside overall performance, reflecting a growing internalization of positional structure as the agent becomes more competent.

7.4 Single Neuron Decoding

Recall that in our linear decoder, each neuron i in the hidden state $h_t \in \mathbb{R}^{512}$ can contribute differently to the predicted position. We define the contribution of neuron i to the prediction of the x-coordinate as:

$$\text{Contribution}_{i,x} \;=\; |A_{1,i}|\,\sigma^{(i)},$$

where $A_{1,i}$ is the corresponding entry in the linear transformation and $\sigma^{(i)}$ is the standard deviation of the hidden state of neuron i over the dataset.

In this analysis, we exclude architectures that include continuous attractor networks (CANs), as their results closely resembled the corresponding non-CAN versions, but with greater noise and less interpretability.

7.4.1 Contribution Distributions

To visualize these contributions, we can plot how the top-contributing neurons vary as a function of Δt, revealing how individual neurons affect predictions at different time offsets into the future. Figure 43 shows five of the most influential neurons from the four models we have trained on the "distance to origin" prediction task.

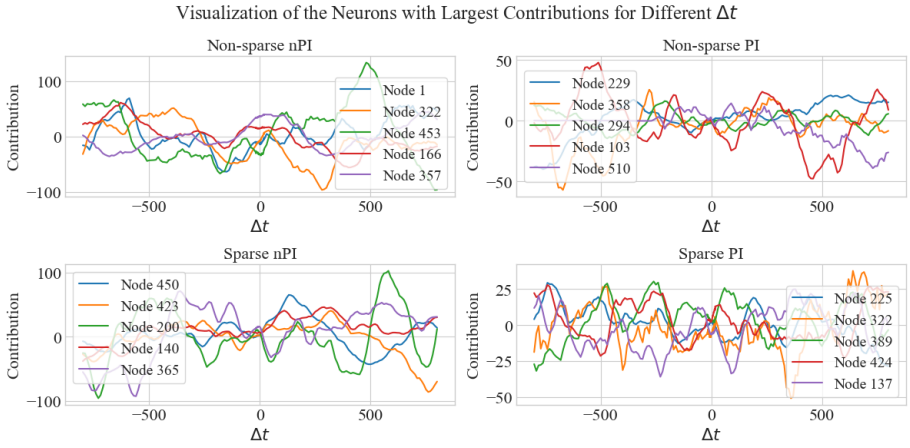


Figure 43: Five high-contribution neurons for distance-to-origin prediction in the four different models we have trained. The contributions are plotted without absolute magnitude so that the behavior remains smooth when crossing the x-axis.

From these plots we can look at only a limited window of predicative models too see how contributions of

single neurons manifests in a certain future prediction. For now, let us look at the models in the range $\Delta t \in (0, 20)$.

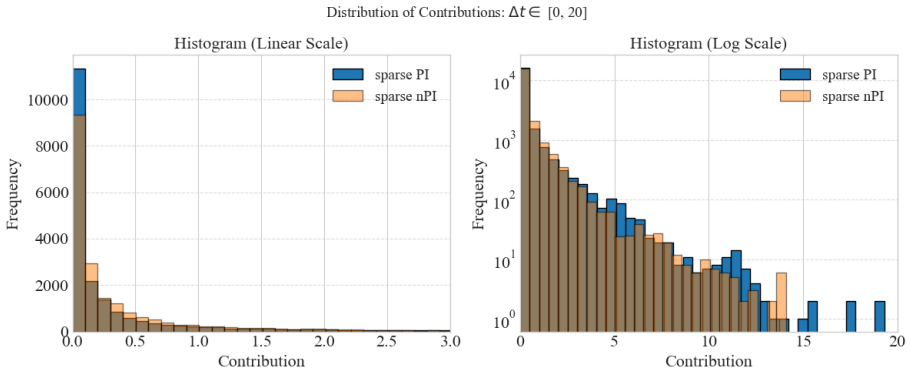


Figure 44: Histogram showing the frequency of contributions for the path-integrating and non-path-integrating agents among all the neurons in the LSTM layer.

We want to examine the distribution of contributions across *all* neurons by plotting histograms of these values (see Figure 44). Most neurons have negligible impact on the decoder's output, suggesting that only a small fraction of the hidden units strongly modulate positional predictions. Notably, in the sparse path-integrating agent, we see both a larger number of neurons with very small contributions *and* a subset of neurons with larger contributions than in the non-path-integrating case.

7.4.2 Top Contributions Comparison

To compare models more directly, we focus on the highest-contributing neurons when predicting near-future time steps ($\Delta t \in [0, 20]$). Figure 45 illustrates that the sparse path-integrating (PI) agent has a handful of neurons that dominate the positional prediction more than those in other architectures. In general, sparse models appear to concentrate the bulk of their predictive capacity in a smaller subset of neurons. This might indicate that the because the sparse network has fewer training parameters it has been **forced to find a more effective representation of space in memory**.

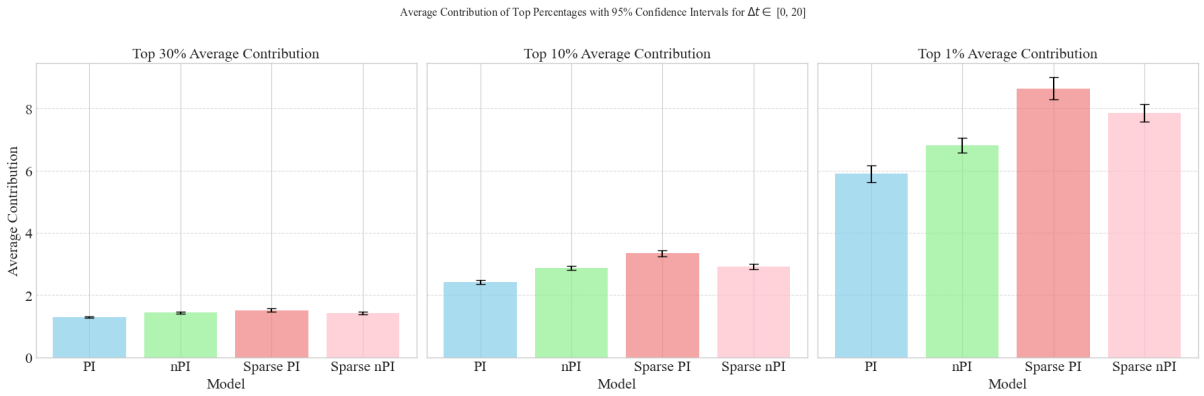


Figure 45: Comparing top neuron contributions across different architectures for near-future predictions ($\Delta t \in [0, 20]$). The sparse PI agent has higher contributions among the top-contributing neurons.

However, we are also interested in the contributions across the entire range of $\Delta t \in (-1000, 1000)$. In Figure 46, we plot the average contributions of the top 1% and top 10% of neurons for all models across different Δt values. The sparse PI architecture exhibits a more pronounced reliance on a small number of high-contribution neurons, especially around near-future and near-past time offsets. This observation suggests that sparse path-integrating models localize spatial encoding into fewer, more specialized neurons.

In summary, these single-neuron analyses reveal two main patterns:

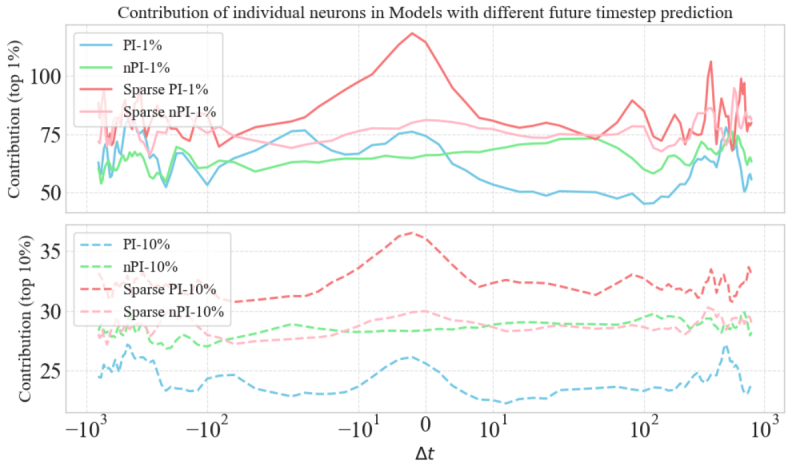


Figure 46: Average contributions of the top 1% and top 10% of neurons, plotted against Δt. The path-integrating agents have bumps near $\Delta t = 0$ indicating that few neurons are very active in short term memory of location.

1. Only a small fraction of neurons significantly influence the position decoder, highlighting the agent's reliance on specialized subsets of hidden units.
2. Sparse path-integrating agents further concentrate these high contributions among fewer neurons, which may indicate a more "compact" representation of spatial information.

7.4.3 RMSE after Removing Top Contributing Neurons

To validate the functional importance of these high-contribution neurons, we perform a lesion analysis by comparing decoding performance under three conditions: using all 512 neurons, using only the top 50 most contributing neurons, and using the bottom 50 least contributing neurons. Here, a neuron's total contribution is defined as the sum of its contributions to both x- and y-coordinate predictions. As shown in Figure 47,

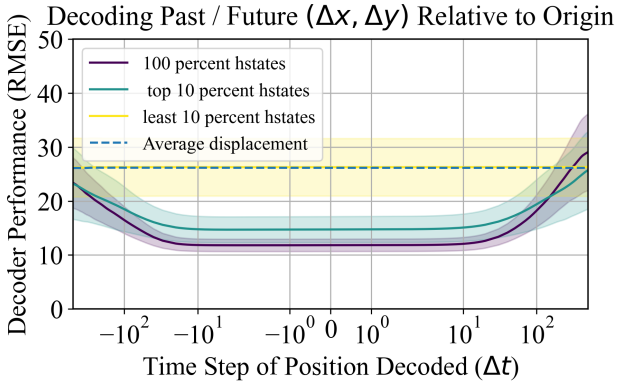


Figure 47: Decoding RMSE using all neurons, the top 50 contributors, and the bottom 50 contributors. The top 10% recover nearly full decoding accuracy, while the bottom 10% perform no better than chance.

the top 10% of neurons almost fully recover the spatial decoding performance of the entire hidden state. In contrast, the bottom 10% contribute no useful information. This striking difference highlights the emergence of a compact, specialized subset of neurons that dominate spatial representation, particularly in the sparse PI architecture.

7.4.4 Coefficient Profiles

To further explore how spatial information is distributed across the hidden state, we examine the coefficient profiles learned by the linear decoder. These coefficients, obtained from the ridge regression model, reflect the relative weight assigned to each neuron when predicting position.

Figure 48 compares the coefficient magnitudes for both sparse and dense path-integrating networks. In the sparse model, we observe fewer pronounced peaks, indicating that only a small subset of neurons are actively used in position decoding. In contrast, the dense model shows a broader spread of moderate contributions across many neurons.

Another notable difference is the apparent overlap between the neurons involved in predicting past and future positions. In the sparse network, the same neurons often contribute to both forward and backward prediction, suggesting that certain units serve as consistent spatial encoders across time. This redundancy is less evident in the dense network, where the coefficients for past and future predictions appear more distinct.

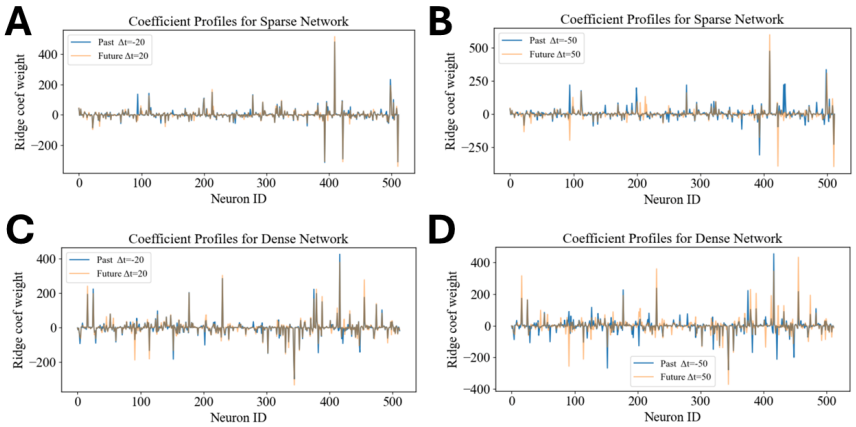


Figure 48: Coefficient profiles from ridge regression for sparse and dense PI networks. The sparse model uses fewer neurons, and shows greater overlap between past and future predictions.

7.5 Grid Cells — Results

7.5.1 Architecture 1

We previously introduced three different strategies for generating velocity scales in Architecture 1. In this section, we evaluate each of them in turn:

- **Fully trainable and independent velocity scales**: Each CAN module learns its own scale without constraint, allowing for maximum flexibility.
- **Trainable geometric scaling (two strategies)**: The velocity scales follow the form $\text{vel}_i = f_0 \cdot r^i$, where we explore two variants—one with r trainable and f_0 fixed, and another with f_0 trainable and r fixed.

Fully trainable and Independent Velocity Scales. Figure 49 illustrates how the learned velocity scales evolve over the course of training. Each CAN is allowed to independently learn its own transformation from action to velocity. Across all networks, we observe a tendency toward high-frequency scaling—significantly larger than what would be biologically plausible. One possible explanation is that the network seeks to amplify movement signals in the bump dynamics, enabling the LSTM to detect and integrate these signals more easily. This may indicate that the network develops a form of "signal-based" movement tracking, rather than establishing a grounded spatial representation.

While this strategy results in active CAN dynamics, it appears to hinder the development of useful spatial codes. Figure 50 shows the firing patterns at the start and end of training for a 96×96 grid-world. In the latter case, the firing patterns lack clear spatial structure. The simultaneous activation of all three CANs

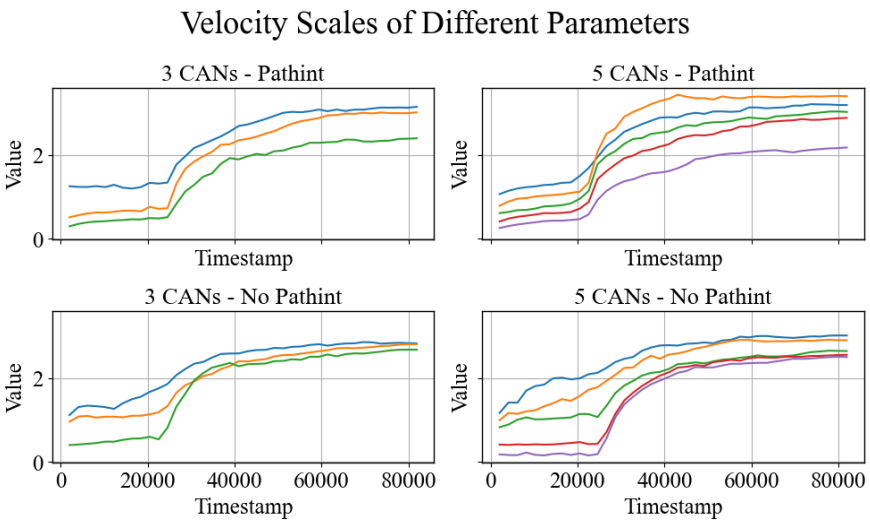
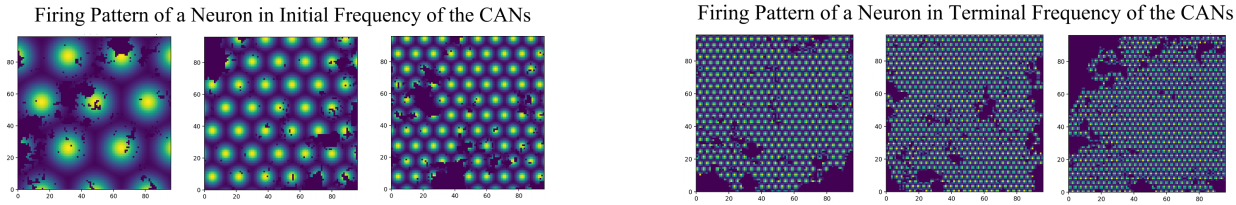


Figure 49: Learned velocity scales throughout training. Frequencies are consistently higher than those observed in biological systems.

across multiple locations implies that the representation is ambiguous and cannot uniquely specify the agent's position.



(a) Early training

(b) Late training

Figure 50: CAN firing patterns early (a) and late (b) in training. No clear spatial structure emerges, and overlap across CANs creates ambiguity.

Despite the fact that the velocity weights are actively used (i.e., non-zero), the scales remain tightly clustered, and the firing activity lacks diversity across CANs. This suggests that the agent utilizes the CANs, but not as a spatial map. Due to the lack of structured encoding, we do not analyze the receptive fields of the LSTM neurons for this condition. The next strategies offer more insight into how different constraints on the velocity scales affect spatial coding.

Trainable Geometric Scales. Next, we examine a constrained version of the velocity scales defined by the geometric form:

$$\text{scale}_i = f_0 \cdot r^i,$$

with $M = 5$ CAN modules. We initialize $f_0 = 0.2$ and $r = 1.5$, and then run two variants: one in which f_0 is trainable and another where r is trainable. Both configurations are tested in sparse agents with and without path integration. The top row of Figure 51 shows how the parameters f_0 or r adapt during training, while the bottom row illustrates how this affects the range of individual velocity scales. As with the fully independent case, the network converges to relatively large frequencies. This suggests a network-wide preference for strong bump displacements, likely reinforcing signal-based movement tracking rather than spatial encoding. The similarity across path-integrating and non-path-integrating agents implies that CANs are not being used in a manner optimized for navigation.

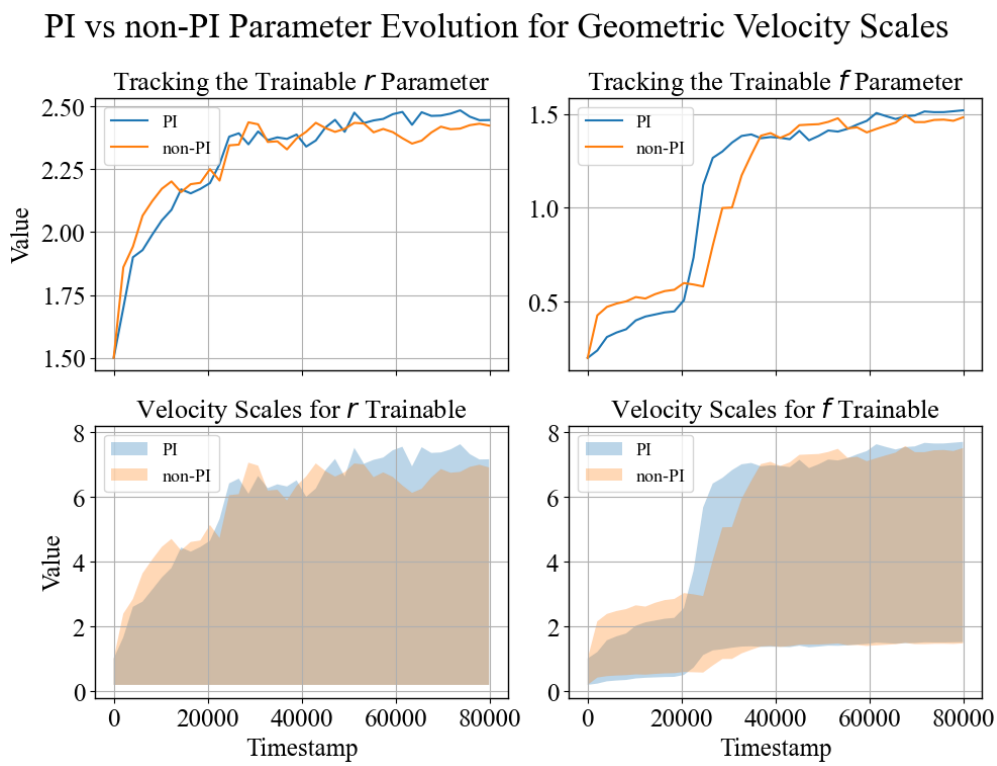


Figure 51: Scales learned throughout training. Both variations learns approximate same distribution for the velocity scales.

To explore whether any spatial structure emerges, we visualize the firing patterns for the path-integrating agent under the "trainable r" configuration (see Figure 52).

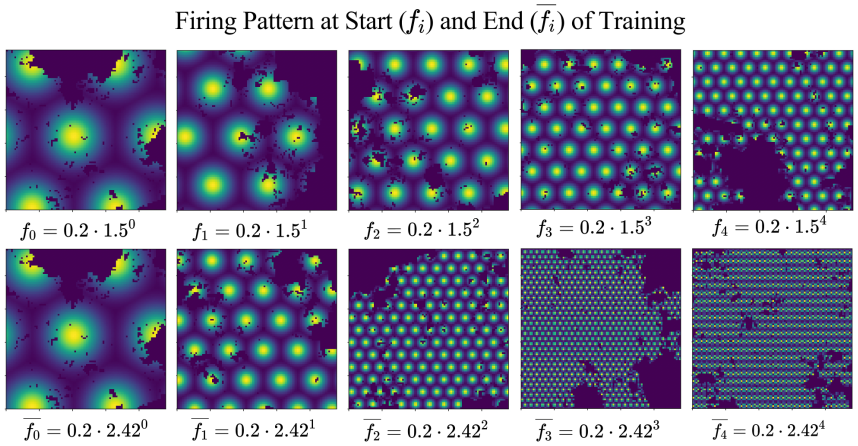


Figure 52: Firing patterns for CAN modules in the path-integrating agent with trainable r. Despite high frequencies in some modules, others show potentially spatially informative activation.

Even though $\overline{f_3}$ and $\overline{f_4}$ operate at frequencies too high for meaningful spatial encoding, we see that $\overline{f_0}, \overline{f_1}$ and $\overline{f_2}$ have firing patterns that could be used as spatial information. To assess this, we compute the Pearson correlations and reconstruct receptive fields between CAN units and LSTM hidden neurons.

First, neuron 379 in the LSTM exhibits the strongest single-unit correlation ($|r| = 0.53$) with one of the neurons in one of the CANs. Its receptive field, shown in Figure 53, reveals a pronounced negative association

with a specific bump location in the first CAN and neglible correlation elsewhere.

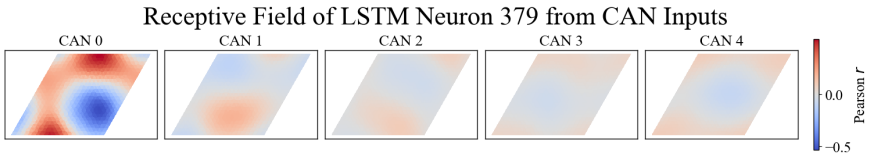


Figure 53: Receptive field of LSTM neuron 379 with respect to all five CAN modules. Only CAN 1 shows significant negative correlation.

We also identify neuron 54, which has the highest sum of absolute correlations across all five CANs. Its receptive field (Figure 54) shows strong, statistically significant interactions with multiple CAN modules—indicating a distributed integration of spatial signals.

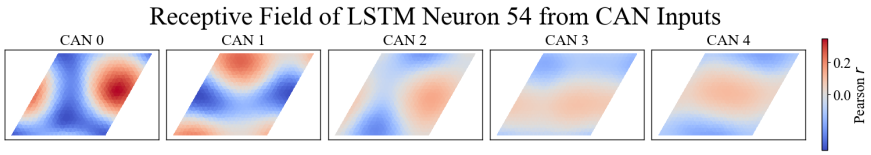


Figure 54: Receptive field of LSTM neuron 54, showing significant coupling with multiple CAN modules—particularly those with both low and high frequencies.

Interestingly, the receptive field patterns resemble first-order Fourier basis functions that encode translation behaviors of the bump. This suggests that, under geometric constraints, the network may recruit CAN outputs in frequency-selective combinations that support navigation-related dynamics rather than explicit spatial encoding.

Conclusion. Across all decoding analyses of Architecture 1, we find that the CANs are not used effectively for spatial encoding. Despite testing various strategies for generating velocity scales—fully trainable and geometrically scaled—we consistently observed that the agent favored high-frequency velocity mappings. These do not correspond well to biologically plausible spatial representations and suggest that the CANs primarily produce transient, high-variance signals rather than stable spatial codes.

Receptive field analyses further reveal that although some LSTM neurons exhibit localized correlations with individual CAN modules, these neurons are not the strongest contributors to position prediction. This supports the hypothesis that the network does not rely on CANs for encoding position. Instead, positional awareness—where it exists—appears to arise independently within the LSTM dynamics.

While some spatial structure was observed in low-frequency CAN modules, these signals were weak and insufficient to drive meaningful decoding performance. Overall, the decoding results indicate that the architectural design of Architecture 1 does not incentivize or enable effective use of the CANs for spatial representation. If such usage exists, it may depend on nonlinear interactions not captured by our current linear decoder. Alternative architectures or loss functions may be required to promote the use of CANs as spatial bases.

7.5.2 Architecture 2

We recall that Architecture 2 implements three different strategies for generating velocities from the hidden state. In this section, we evaluate each approach in turn:

- **Trainable nonlinearity:** $v_t = \tanh(Wh_t)$, where the weight matrix W is learned and then squished into a reasonable interval $(-1, 1)$.
- **Trainable linear mapping:** $v_t = Wh_t$, where W is a fully trainable weight matrix.
- **Fixed random mapping:** $v_t = Wh_t$, where W is randomly initialized and held constant throughout training.

Trainable Nonlinearity. We begin by examining the behavior of the velocity-generating mechanism in the architecture where the weight is trainable and then the tanh activation function is applied. The resulting velocities v_t are effectively constant over time, taking values close to the extremes $(\pm 1, \pm 1)$ — the saturation limits of the tanh function. This indicates that the network has learned to output static, maximal velocities, thereby ignoring any meaningful temporal information. In practice, this renders the CANs funcionally useless, as each bump moves uniformly in a fixed direction and speed across all episodes.

Since the CANs are always initialized at the same position at the start of each episode, the only information they may encode is the initial timestep. This is reflected in the receptive field analysis: only one LSTM neuron (ID 450) shows a statistically significant correlation with the CANs, peaking at 0.146. The receptive field of this neuron appears to track the initial bump position, potentially enabling the network to establish the fixed output velocity at the start. All other hidden states exhibit negligible correlations.

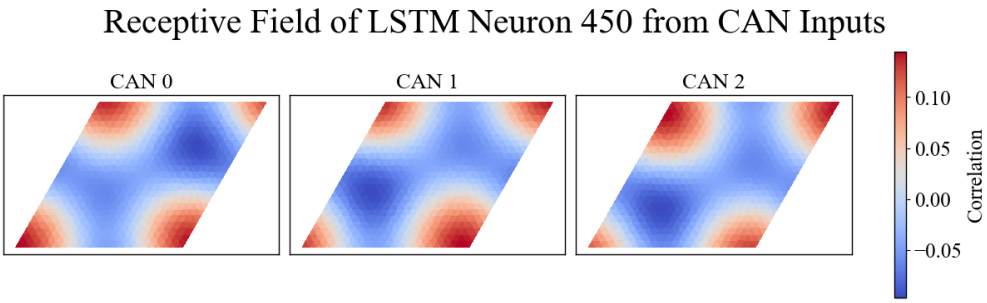


Figure 55: Receptive field of the most strongly CAN-correlated LSTM neuron (ID 450), indicating weak overall influence of the CANs.

In summary, the use of a trainable tanh activation allows the network to trivially suppress the CANs by saturating their outputs. This results in fixed, non-informative velocity signals and minimal interaction between the CANs and the rest of the architecture. To encourage meaningful use of the CAN dynamics, we next examine architectures without a nonlinearity, where the network cannot rely on saturation and must instead learn to utilize the CANs more actively.

Trainable Linear Mapping. We now examine the setup where the velocity vector v_t is produced by a trainable linear transformation of the hidden state: $v_t = Wh_t$, without any nonlinearity. For this analysis, we focus on the path-integrating agent using three CANs.

As illustrated in Figure 56, the resulting velocities are exceptionally large. Many component-wise values exceed ± 20, meaning that the bumps in the CANs could theoretically rotate more than once around the toroidal space in a single timestep. This magnitude is conceptually problematic: if the bump completes multiple wraps per step, its position becomes ambiguous, undermining its role as a spatial code.

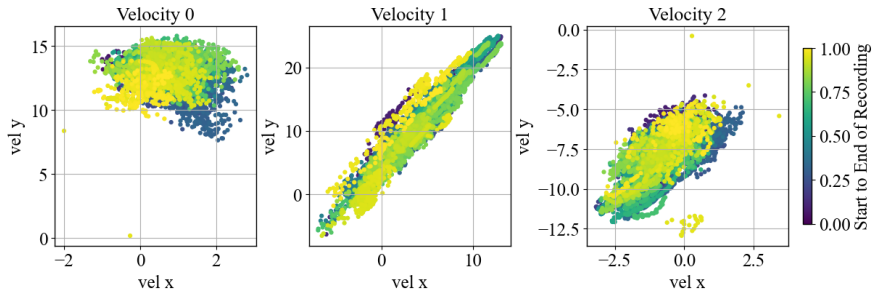


Figure 56: Velocity outputs during training with a trainable linear mapping. Several components exceed a magnitude of 20, causing bumps to rotate multiple times around the torus.

We summarize the statistics of these velocities in the table below. As seen, CAN1 exhibits the highest average magnitudes and variability:

	Mean $(v_x, v_y y)$	**SD** (v_x, v_y)
CAN0	$(0.787,\ 12.955)$	$(0.659,\ 1.260)$
CAN1	$(4.933,\ 12.256)$	$(3.842,\ 6.083)$
CAN2	$(-0.924,\ -8.298)$	$(0.915,\ 1.592)$

Table 4: Velocity statistics for each CAN, showing directional mean and variability.

To assess whether the CANs encode meaningful environmental signals, we computed correlations between CAN activity and different environmental features. As shown in Figure 57, the strongest (although weak) correlations were found with distance to skeletons. However, these correlations barely reach significance and may not reflect meaningful use of the CAN signals by the agent.

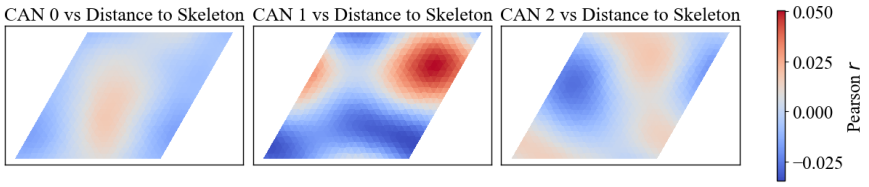


Figure 57: Weak correlations between CAN outputs and environmental variables, with the strongest signals related to skeleton structure.

To further understand the bump behavior, we analyzed accelerations—defined as the change in velocity across timesteps. The statistics, shown below, indicate that despite the large instantaneous velocities, bump accelerations are moderate and comparable across CANs. This suggests that while velocities are extreme, they may be relatively stable across time.

	Mean (a_x, a_y)	**SD** (a_x, a_y)
CAN0	$(0.141,\ 0.232)$	$(0.177,\ 0.324)$
CAN1	$(0.365,\ 0.579)$	$(0.535,\ 0.867)$
CAN2	$(0.154,\ 0.235)$	$(0.199,\ 0.332)$

Table 5: Acceleration statistics per CAN, showing stable dynamics despite large velocities.

In summary, while this architecture allows for richer interactions between the hidden state and CANs compared to the saturating tanh case, the learned velocities are still implausibly large. This likely limits the CANs' utility for encoding spatial information, as repeated toroidal wrapping undermines location-specific activity. The weak correlations with environmental variables further suggest that the network is not leveraging the CANs in a meaningful or interpretable way.

Fixed Random Mapping In this configuration, the velocity vector $v_t = Wh_t$ is determined by a fixed, randomly initialized weight matrix W. We analyze this setup in a path-integrating agent using three CANs.

Figure 58 shows the velocity profiles over time. Unlike previous architectures, the velocities here remain within a moderate range, with maximum values around 2.5. This magnitude falls within a biologically plausible regime and aligns with the upper bound required for generating structured grid-like (e.g., hexagonal) firing patterns.

The summary statistics in the Table 6 further support this stability:

Despite the promising range and variance, an important limitation is that each velocity component rarely changes sign. For instance, CAN0's v_y component component remains consistently negative, indicating that the bump effectively rotates in a single direction around the torus throughout an episode. This severely restricts the expressiveness of the CAN and may stem from the fixed linear mapping lacking sufficient com-

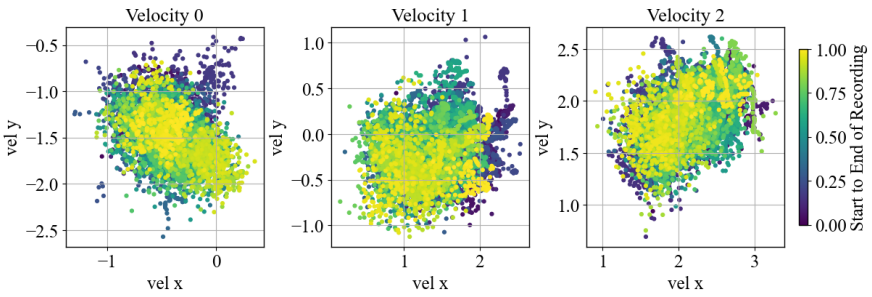


Figure 58: Velocity outputs over time with fixed random weight mapping.

	Mean (v_x, v_y)	**SD** (v_x, v_y)
CAN0	$(-0.403, -1.466)$	$(0.240, 0.285)$
CAN1	$(1.442, -0.166)$	$(0.449, 0.332)$
CAN2	$(2.150, 1.861)$	$(0.383, 0.282)$

Table 6: Velocity statistics per CAN. The values suggest moderate and consistent motion.

plexity. Adding a nonlinear transformation or deeper projection layer could introduce more diverse bump dynamics.

To better understand the bump behavior, we examined velocity changes across time (i.e., acceleration). The results in Figure 59 and the accompanying table reveal modest variation, further indicating that the CANs operate in a relatively stable, albeit constrained, regime.

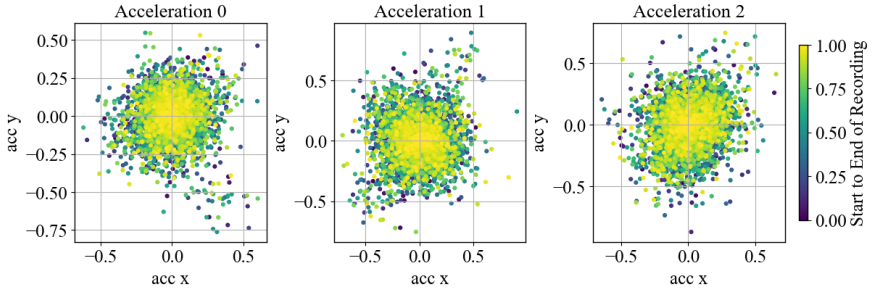


Figure 59: Acceleration over time. The magnitudes remain modest, consistent with stable bump motion.

	Mean $abs(a_x, a_y)$	**SD** $abs(a_x, a_y)$
CAN0	$(0.066, 0.068)$	$(0.081, 0.083)$
CAN1	$(0.091, 0.084)$	$(0.107, 0.107)$
CAN2	$(0.076, 0.089)$	$(0.089, 0.105)$

Table 7: Acceleration statistics per CAN (values shown as (x, y) points).

To assess functional relevance, we correlated the CAN activity with high-level behavioral variables. As shown in Figure 60, the only significant relationship was found with the distance to the ranged predator (i.e., skeleton with bow and arrow). Although this correlation is weak, it indicates that the network has learned to encode some task-relevant spatial variable via the CANs.

Finally, we examined how CAN activity influences LSTM neurons. The neuron with the strongest receptive field—unit 457—shows a significant localized correlation, consistent with encoding some spatial information linked to bump location or environmental cues.

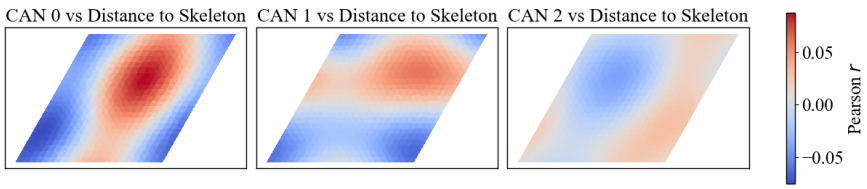


Figure 60: Correlation between CAN activity and distance to the ranged predator.

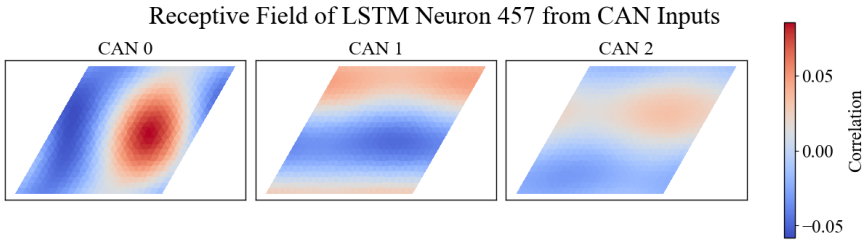


Figure 61: Receptive field of neuron 457 from CAN input. The localized structure suggests selective sensitivity to bump location.

This receptive field bears similarity to the correlation map with the predator's distance, further hinting at shared encoding structure. Despite being limited by fixed weights, the CANs in this setup show mild task relevance and manageable dynamics—making this architecture a more interpretable baseline for future comparisons.

Conclusion. In summary, the trainable nonlinearity variant failed to make effective use of the CANs, as their activity was saturated at the limits of the tanh function, rendering them essentially inactive. The trainable linear mapping may have captured some information about enemy position, but the velocity magnitudes were far too large to encode meaningful information in a traditional sense. The fixed random mapping showed more reasonable velocity scales and demonstrated some promise, particularly in encoding predator distance, although it too struggled with expressiveness.

Overall, enabling the velocity signal to carry useful information remains a key challenge. One potential solution is to introduce a more flexible transformation—such as an additional layer—between the LSTM and the velocity output. This could help the agent represent richer information more clearly and potentially improve task performance. Exploring such architectural modifications would be a valuable direction for future work.

8 Conclusion

In this thesis, we investigated how architectural variations—namely path integration, connectivity sparsity, and the inclusion of CANs — influence spatial reasoning, memory, and adaptive behavior in reinforcement learning agents trained with clipped PPO. All agents successfully learned to navigate a dynamic environment, locating food patches and evading predators. Among these factors, the inclusion of a path-integrating module was the strongest predictor of high survival time, whereas sparsity had little effect on performance within the path-integrating condition. This highlights the utility of sparse representations: they allow for efficient computation while preserving or even enhancing functional capability, much like neural systems in biology.

Behavioral analyses revealed nuanced strategies shaped by internal state representations. Agents with path integration ventured significantly farther from the origin, suggesting enhanced exploratory confidence enabled by richer spatial encoding. Post-simulation statistics showed that the likelihood of revisiting a previously visited food patch decreased with higher recent consumption ("eat-rate") and shorter distance to the patch. However, when positional uncertainty was high—i.e., when the agent had weaker internal confidence in its location—it became more likely to return to familiar patches.

Sparse path-integrating agents exhibited the strongest positional awareness, achieving nearly the same predictive accuracy using only the top 10% most contributive neurons—demonstrating notable neural efficiency. While these agents lacked a consistent encoding of relative position, there were indications that they retained some memory of the direction they had come from. These findings reinforce the notion that targeted connectivity and specialized neuronal subsets can support robust spatial representations with minimal computational overhead.

We derived both numerical and analytical CAN models from mathematical first principles, ensuring biologically plausible and stable bump dynamics. These models successfully produced structured grid-cell-like activity, yet their incorporation into agent architectures had minimal effect on overall task performance. This suggests current architectural pathways may not effectively leverage the CANs' structured representations. Interestingly, however, post hoc analysis revealed that CANs were not entirely ignored. In some configurations, they correlated modestly with latent environmental variables—particularly the distance to ranged predators—hinting at their potential to encode abstract "threat spaces." In one architecture, specific LSTM neurons even exhibited receptive fields aligned with localized bump activity, suggesting limited but targeted use of CAN input. These findings point to promising directions for future work in designing architectures that can more effectively harness CAN-driven representations.

9 Future Work

Although our initial findings are promising, several avenues remain for extending and refining this work:

Exploring More Advanced Architectures. The results suggest that feeding CAN outputs directly into the LSTM layer may have introduced noise, potentially hindering learning. The LSTM appeared to allocate its memory capacity selectively, often prioritizing other forms of task-relevant information over detailed spatial encoding. Future research could explore alternative integration strategies—for instance, feeding CAN representations directly into the actor-critic networks, either in parallel with or in place of the LSTM inputs. Another promising direction would be to insert an intermediate processing layer between the CANs and the LSTM. This layer could serve to refine or compress spatial signals, potentially making it easier for the LSTM to store and utilize positional information effectively.

Developing a More Advanced Decoder. Thus far, position decoding has relied solely on a linear model, which prioritizes interpretability but may overlook non-linear interactions within the network. Given the observed correlations between CAN outputs and LSTM activity, it is possible that the LSTM encodes spatial information in a more distributed or indirect manner—potentially through multi-layer transformations. Employing a non-linear decoder, such as a small feedforward neural network, could help uncover whether positional information from the CANs is being utilized in a less direct fashion. While this approach would sacrifice some interpretability, it could provide deeper insight into how spatial signals are integrated and transformed within the agent's architecture.

Numerical CANs Without Explicit Velocity Inputs. Thus far, the CANs have been driven exclusively by velocity-based inputs, reflecting their traditional use for path integration. However, expanding the connectivity to allow all-to-all input from upstream network layers presents a promising direction. This would enable the CANs to respond to a broader range of signals, potentially encoding more abstract or high-level variables. Indeed, we already observed that the CANs showed modest sensitivity to predator distance—suggesting they can represent more than physical location. With richer input connectivity, the CANs could develop localized activity patterns based on environmental context, enabling more nuanced spatial or semantic representations within their neural sheet.

Additional Decoding Objectives. While this thesis focused on decoding absolute and relative position, other spatial variables may offer complementary insights. One promising avenue is the prediction of relative angle or heading direction. This target is simpler and may reveal whether the agent maintains a sense of orientation, even when its encoding of distance or displacement is limited. Exploring such signals could help disentangle different components of spatial awareness and uncover latent structure in the hidden-state representations.

In summary, the proposed research directions aim to deepen our understanding of how artificial agents represent and leverage spatial information. By drawing inspiration from biological navigation systems and exploring more refined architectures, decoders, and connectivity schemes, we can move toward models that are not only more effective computationally but also more aligned with the principles of neural computation observed in nature.

References

[1] John O'Keefe and Lynn Nadel. *The Hippocampus as a Cognitive Map*. Oxford University Press, Oxford, UK, 1978. ISBN 9780198572060.

[2] Hamid R. Tizhoosh. Reinforcement learning based on actions and opposite actions. *AIML Conference Proceedings*, 2005.

[3] Xiaoyun Lei, Zhian Zhang, and Peifang Dong. Dynamic path planning of unknown environment based on deep reinforcement learning. *Journal of Robotics*, 2018:Article ID 5781591, 2018. doi: 10.1155/2018/5781591.

[4] Maciej Grzelczak and Piotr Duch. Deep reinforcement learning algorithms for path planning domain in grid-like environment. *Applied Sciences*, 11(23):11335, 2021. doi: 10.3390/app112311335.

[5] Phone Thiha Kyaw, Aung Paing, Theint Theint Thu, Rajesh Elara Mohan, Anh Vu Le, and Prabakaran Veerajagadheswar. Coverage path planning for decomposition reconfigurable grid-maps using deep reinforcement learning based travelling salesman problem. *IEEE Access*, 8:225945–225957, 2020. doi: 10.1109/ACCESS.2020.3045027.

[6] YungMin SunWoo and WonChang Lee. Comparison of deep reinforcement learning algorithms: Path search in grid world. In *International Conference on Electronics, Information, and Communication (ICEIC)*, pages 20–21. IEEE, 2021. doi: 10.1109/ICEIC51217.2021.9369800.

[7] Hamid R. Tizhoosh. Reinforcement learning based on actions and opposite actions. *AIML Conference Proceedings*, 2005.

[8] S Hochreiter. Long short-term memory. *Neural Computation MIT-Press*, 1997.

[9] W. Xiong, L. Wu, F. Alleva, J. Droppo, X. Huang, and A. Stolcke. The microsoft 2017 conversational speech recognition system. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5934–5938, 2018. doi: 10.1109/ICASSP.2018.8461870.

[10] OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation, 2019. URL `https://arxiv.org/abs/1808.00177`.

[11] Maximilian Beck, Korbinian Pöppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. xlstm: Extended long short-term memory, 2024. URL `https://arxiv.org/abs/2405.04517`.

[12] Torkel Hafting, Marianne Fyhn, Sturla Molden, May-Britt Moser, and Edvard I. Moser. Microstructure of a spatial map in the entorhinal cortex. *Nature*, 436(7052):801–806, 2005. doi: 10.1038/nature03721.

[13] Yoram Burak and Ila R. Fiete. Accurate path integration in continuous attractor network models of grid cells. *PLoS Computational Biology*, 5(2):e1000291, 2009. doi: 10.1371/journal.pcbi.1000291.

[14] Bartók Ságodi, Szabolcs Káli, and Ila R. Fiete. Robust grid-cell attractor dynamics in conductance-based spiking neural networks. *eLife*, 13:eXXXXXX, 2024. In press.

[15] Christopher J. Cueva and Xue-Xin Wei. Emergence of grid-like representations by training recurrent neural networks to perform spatial localization. In *Proc. International Conference on Learning Representations (ICLR)*, 2018. URL `https://openreview.net/forum?id=BkwzxnC9FX`.

[16] Andrea Banino, Caswell Barry, Piotr Mirowski, Charles Blundell, et al. Vector-based navigation using grid-like representations in artificial agents. *Nature*, 557:429–433, 2018. doi: 10.1038/s41586-018-0102-6.

[17] Benjamin Ellis Mikayel Samvelyan Matthew Jackson Samuel Coward Jakob Foerster Michael Matthews, Michael Beukman. Craftax: A lightning-fast benchmark for open-ended reinforcement learning. 2, 2024. URL `https://arxiv.org/abs/2402.16801#`.

[18] Danijar Hafner. Benchmarking the spectrum of agent capabilities. In *International Conference on Learning Representations (ICLR)*, 2022. URL `https://openreview.net/forum?id=1W0z96MFEoH`.

[19] Felix Baastad Berg. Spatial representation in long short term memory for for- aging agents trained with proximal policy optimization. *TMA4500*, 2025.

[20] Caron Lab. Mushroom body. *The Caron Lab*, n.d. URL https://www.thecaronlab.com/mushroombody.

[21] Glenn C. Turner and Joshua T. Vogelstein. A connectome of the drosophila central complex reveals network heterogeneity. *Nature Communications*, 13:2022, 2022. doi: 10.1038/s41467-022-29613-5.

[22] Ryan Badman. Forageworld: Rl agents in complex foraging arenas develop internal maps for navigation and planning. Seminar at Lyon Neuroscience Research Centre, December 2024. URL https://www.crnl.fr/en/event/seminaire-ryan-badman-forageworld-rl-agents-complex-foraging-arenas-develop-internal-maps.

[23] Riley Simmons-Edler, Ryan P. Badman, Felix Baastad Berg, Raymond Chua, John J. Vastola, Joshua Lunger, William Qian, and Kanaka Rajan. Deep rl needs deep behavior analysis: Exploring implicit planning by model-free agents in open-ended environments. arXiv:2506.06981 [cs.AI], 2025.

[24] Anonymous. Policy iteration in rl: An illustration, 2023.

[25] Prafulla Dhariwal Alec Radford Oleg Klimov John Schulman, Filip Wolski. Proximal policy optimization algorithmss. 2017. URL https://arxiv.org/pdf/1707.06347.

[26] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015. https://arxiv.org/pdf/1506.02438.

[27] Andrea Banino, Caswell Barry, Benigno Uria, Charles Blundell, Timothy Lillicrap, Piotr Mirowski, Alexander Pritzel, Martin J Chadwick, Thomas Degris, Joseph Modayil, Greg Wayne, Hubert Soyer, Fabio Viola, Brian Zhang, Ross Goroshin, Neil Rabinowitz, Razvan Pascanu, Charlie Beattie, Stig Petersen, Amir Sadik, Stephen Gaffney, Helen King, Koray Kavukcuoglu, Demis Hassabis, Raia Hadsell, and Dharshan Kumaran. Vector-based navigation using grid-like representations in artificial agents. *Nature*, 557(7705):429–433, 2018. doi: 10.1038/s41586-018-0102-6. URL https://pubmed.ncbi.nlm.nih.gov/29743670/.

[28] John O'Keefe and Jonathan Dostrovsky. The hippocampus as a spatial map. preliminary evidence from unit activity in the freely-moving rat. *Brain Research*, 34(1):171–175, 1971. doi: 10.1016/0006-8993(71)90358-1. URL https://www.sciencedirect.com/science/article/pii/0006899371903581.

[29] Edvard I. Moser, Emilio Kropff, and May-Britt Moser. Place cells, grid cells, and the brain's spatial representation system. *Annual Review of Neuroscience*, 31:69–89, 2008. doi: 10.1146/annurev.neuro.31.061307.090723. URL https://pubmed.ncbi.nlm.nih.gov/18284371/.

[30] Kanstantsin Bohté and Peter J Kindermans. Grid-like coding increases the capacity of continuous attractor networks. In *Advances in Neural Information Processing Systems*, volume 26, 2013. URL https://proceedings.neurips.cc/paper_files/paper/2013/hash/46c3b46f992f4e64c27b327e1f3e6f11-Abstract.html.

[31] Sarthak Chandra, Sugandha Sharma, Rishidev Chaudhuri, and Ila Fiete. Episodic and associative memory from spatial scaffolds in the hippocampus. *Nature*, 638:739–751, 2025. doi: 10.1038/s41586-024-08392-y. URL https://www.nature.com/articles/s41586-024-08392-y.

[32] Benjamin Groh and Ila R Fiete. A grid cell code for ego-centric space. *Journal of Neuroscience*, 38 (40):8533–8540, 2018. doi: 10.1523/JNEUROSCI.2277-17.2018. URL https://fietelab.mit.edu/wp-content/uploads/2018/12/gridcell_jneurosci-1.pdf.

[33] Hanne Stensola, Tor Stensola, Trygve Solstad, Kristian Frøland, May-Britt Moser, and Edvard I. Moser. The entorhinal grid map is discretized. *Nature*, 492(7427):72–78, December 2012. doi: 10.1038/nature11649.

[34] Alexander Mathis, Andreas V. M. Herz, and Martin Stemmler. Optimal population codes for space: Grid cells outperform place cells. *Neural Computation*, 24(9):2280–2317, September 2012. doi: 10.1162/NECO_a_00319.

[35] Moritz Hardt, Benjamin Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. 2016. URL `https://arxiv.org/abs/1509.01240`.

[36] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[37] Henrik Mouritsen. Long-distance navigation and magnetoreception in migratory animals. *Nature*, 558: 50–59, 2018. doi: 10.1038/s41586-018-0176-1.

A Appendix - Lyon Poster

Poster presented at the Lyon Neuroscience Research Centre.

Neural mechanisms of planning and memory in dynamic foraging agents

Ryan P. Badman* [1,2], Riley Simmons-Edler* [1,2], Felix Berg [2], Joshua Lunger [3], John Vastola [1], William Qian [2], Kanaka Rajan [1,2]

[1]Neurobiology Department, Harvard Medical School; [2]Kempner Institute, Harvard University; [3]Computer Science Department, University of Toronto * equal contributions

INTRODUCTION[1-4]

- We have little understanding of how biological spatial navigation, planning, & memory work outside of small, usually fully-observable arenas.
- We don't even know how standard deep RL agents do navigation, despite 100s of ML benchmarks that require memory and planning in 2D and 3D worlds.
- **GOAL: Use partially-observable, simulated foraging arenas to develop methods and frameworks to analyze continuous naturalistic data in neuroscience**

RESEARCH QUESTIONS

Is simple memory and reinforcement learning sufficient to navigate in large, partially observable foraging arenas?

- *Mammals and insects are known to keep track of relative position and orientation (from an arbitrary starting point) by integration of self-motion tracking and visual cues.*
- *Here proximal policy optimization (PPO) and long short-term memory (LSTM) are used for RL and memory.*

If so, can high performance be maintained for connectome-informed sparse networks?

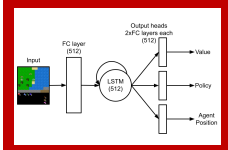
- *Neural networks in theory sub-fields are typically highly connected, while real brains are sparse [5].*

What are the neural circuits underlying goal-oriented navigation, memory and planning in large, complex spaces?

- *Neural and behavioral logging during task performance will allow GLMs and manifold studies.*

NEURAL NETWORK ARCHITECTURE

We used a standard, state-of the art deep learning network architecture for reinforcement learning (PPO) with recurrent memory (LSTM). PPO is a policy gradient method that uses a value (advantage) function. LSTM is a recurrent neural network that allows gating between recurrent neural units and excels at learning multi-timescale relationships in tasks.



Schematic of network architecture used in our study.

We were able to achieve comparable performance between highly sparse (90% zero weights) vs highly connected networks, and with or without a path integration objective loss term.

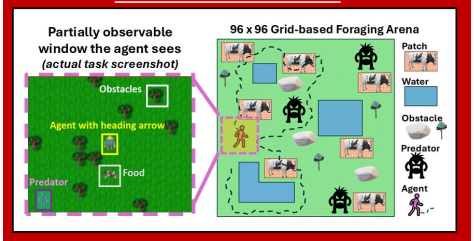
[6] Craftax

Kempner INSTITUTE

UNIVERSITY OF TORONTO

HARVARD MEDICAL SCHOOL

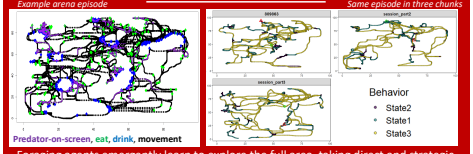
FORAGING TASK DESIGN [6]



Task Description

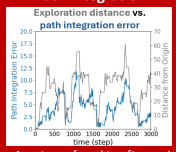
- We heavily modified the Craftax task [6] to be suitable for neuro research.
- In each episode a new random arena is generated so the agent is never overtrained on a given environmental configuration.
- The agent's goal is **survival**, i.e. to reduce its steadily increasing hunger, thirst, and fatigue levels by eating, drinking, and sleeping as needed.
- The input is a small 7 x 9 partially observable window, centered on the agent as it moves around a 96 x 96 tile arena.
- Food is cows which move around, water is provided by lakes, and sleep locations must be chosen based on identifying areas with less predators.
- During the task behavior and neural activity are logged at every time step.

BEHAVIOR RESULTS



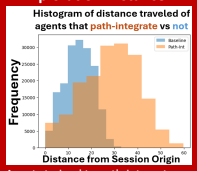
Foraging agents emergently learn to explore the full area, taking direct and strategic routes to discovered patches in memory. Bayesian path segmentation is used to identify movement states that are alternated between, analyzed below.

Path Integration



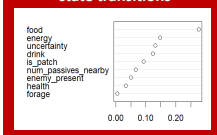
Agents are found to often revisit the origin to restore path integration performance, but can predict their position with about 5-tile precision overall.

Exploration Distance



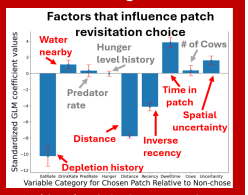
Agents trained to path-integrate explore farther than those not trained to, despite overall performance being similar.

Variables driving movement state transitions



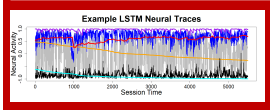
Short-, mid-, and long-range movement patterns are modulated by a combination of task variables. Factors in the plot are unsigned decision tree importance scores.

Patch-revisiting choice GLM



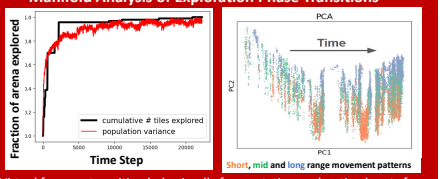
Multi-goal optimization emerges in agents. One of the goals is revisiting patches where uncertainty was higher.

MODEL-GENERATED NEURAL ACTIVITY RESULTS

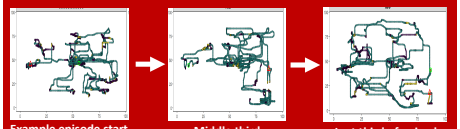


Recurrent activity is generated by 512 RNN (LSTM) units that are connected to the RL layers. All weights in the network are trainable. Neural activity is read out at every time step.

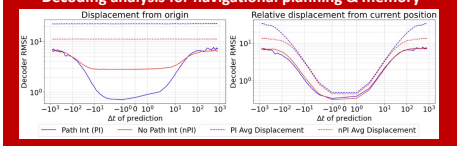
Manifold Analysis of Exploration Phase Transitions



Virtual foragers transition behaviorally from rotating exploration bursts from the origin, to wider loops through discovered patches. We see neural transitions as well in population variance and PCA space.



Decoding analysis for navigational planning & memory



Path integrating agents have improved decoding performance for predicting past and future positions from the LSTM recurrent neurons. The planning and memory time horizons are on the order of 100s of time steps,

CONCLUSIONS & NEXT STEPS

- Virtual foraging agents, trained with standard RL and memory architectures, performed well in large foraging arenas.
- In each new random arena, agents switched from early exploration phases to strategic and direct revisitation of patches in memory.
- Agents could be trained to predict their distance from an episode's origin, which also allowed a readout of the agent's spatial uncertainty at each time step.
- Furthermore, agents could be trained with brain-like sparsity, with 90% of the network weights zeroed, without losing task performance.
- Being trained to also path-integrate led to farther exploration and more decodable neural signals of planning and memory.
- Next steps include finishing current analyses, and exploring grid cell-augmented RNN architectures to better connect to mammalian studies.

REFERENCES

[1] Wolbers, T. Et al. "Challenges for identifying the neural mechanisms that support spatial navigation: the impact of spatial scale." Frontiers in human neuroscience 8 (2014): 571
[2] Wen, John H., et al. "One-shot entorhinal maps enable flexible navigation in novel environments." Nature (2024): 1-8.
[3] Wehner, R. "Searching behaviour of desert ants, genus Cataglyphis (Formicidae, Hymenoptera)." Journal of comparative physiology 142 (1981): 315-338.
[4] Osborne, Juliet L., et al. "The ontogeny of bumblebee flight trajectories: from naïve explorers to experienced foragers." Plos one 8.11 (2013): e78681.
[5] Broido, Anna D., and Aaron Clauset. "Scale-free networks are rare." Nature communications 10.1 (2019): 1017
[6] Matthews, Michael, et al. "Craftax: A Lightning-Fast Benchmark for Open-Ended Reinforcement Learning." arXiv preprint arXiv:2402.16801 (2024).

B Appendix - Lyon talk

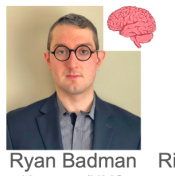
Seminar held by Ryan Badman 17th December at the Lyon Neuroscience Research Centre.

ForageWorld: RL agents in complex foraging arenas develop internal maps for navigation and planning

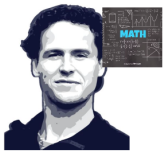
Ryan Badman, PhD
Research Associate
Harvard Medical School & Kempner Institute
17/12/2024

- Leonard and Isabelle Goldenson Fellowship
- Lefler Neurodegeneration Grant

Project Group: *healthy collaboration of CS and neuro researchers*


Ryan Badman
Kempner/HMS

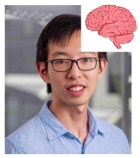

Riley Simmons-Edler
Kempner/HMS



Felix Berg
MIT/NTNU


John Vastola
HMS


Joshua Lunger
U. Toronto


William Qian
Kempner/Harvard


Kanaka Rajan
Kempner/HMS

Space and Memory: two big knowledge gaps in neuroscience

- **Space:** Most real-world decisions occur in large partially-observable spaces (physical or conceptual). Most neuroscience lab tasks do not.
- **Memory:** We have poor understanding of long-term memory structure and recall.
- Good future planning requires accurate space-feature maps and recall. Neuro models and AI/ML models are both bad at this.

Beyond Trial-Based Paradigms: Continuous Behavior, Ongoing Neural Activity, and Natural Stimuli

Alexander Huk, Kathryn Bonnen, and Biyu J. He

Article | Open access | Published: 05 March 2024

Population coding of strategic variables during foraging in freely moving macaques

Neda Shahidi, Melissa Franch, Arun Parajuli, Paul Schrater, Anthony Wright, Xaq Pitkow & Valentin Dragoi

Nature Neuroscience 27, 772–781 (2024) | Cite this article

Keep it real: rethinking the primacy of experimental control in cognitive neuroscience

Samuel A. Nastase, Ariel Goldstein, Uri Hasson

Behav Neurosci. 2021 Oct;135(5):601-609. doi: 10.1037/bne0000471. Epub 2021 Jun 7.

The primacy of behavioral research for understanding the brain

Yael Niv

Emphasis on more continuous and/or naturalistic neuroscience tasks, but are our analysis frameworks ready for it?

DeepMind

Open-Ended Learning Leads to Generally Capable Agents

Open-Ended Learning Team*, Adam Stooke, Anuj Mahajan, Catarina Barros, Charlie Deck, Jakob Bauer, Jakub Sygnowski, Maja Trebacz, Max Jaderberg, Michael Mathieu, Nat McAleese, Nathalie Bradley-Schmieg, Nathaniel Wong, Nicolas Porcel, Roberta Raileanu, Steph Hughes-Fitt, Valentin Dalibard and Wojciech Marian Czarnecki

DeepMind, London, UK

A parallel push in computer science towards "open-ended" exploration tasks to discover more intelligent reinforcement learning computations.

Give a very general end-goal, but the agent needs to create and accomplish its own chain of subgoals to reach the end-goal.

More complex simulated worlds are important to develop methods and frameworks to analyze naturalistic data in neuroscience

- We have little understanding of how spatial navigation, planning & memory work (outside of small, fully-observable arenas) in animals or insects.
- We don't know how standard deep RL agents do navigation, despite 100s of ML benchmarks that require memory and planning in 2D & 3D worlds.
- More complex data needs more statistics to analyze properly → simulation
- Here, we *combine the strengths of ML simulations with the depth of neuroscience analyses* to prepare for the naturalistic future of our field.

Wolbers, Thomas, and Jan M. Wiener. "Challenges for identifying the neural mechanisms that support spatial navigation: the impact of spatial scale." *Frontiers in human neuroscience* 8 (2014): 571.

Deep learning networks are smaller but comparable to insect brain sizes

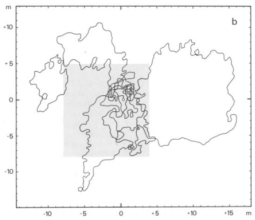
Growing recognition that *individual* insects have rich behavioral dynamics (in addition to collective intelligence)

COSYNE 2024
Keynote

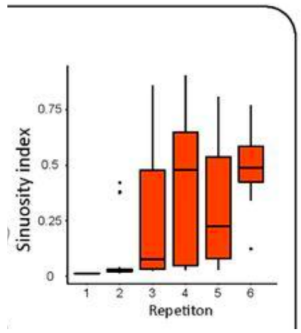
Example 1: Desert ants' spatial memory and strategy

- If a homing ant (Cataglyphis bicolor,C. albicans) gets lost, it does not perform a random walk but adopts a stereotyped search strategy.
- During its search the ant performs a number of loops of ever-increasing size, starting and ending at the origin and pointing at different azimuthal directions.
- After one hour of continuous search the ant's search paths cover an area of about 104 m^2, precisely centred around the origin.

Wehner, Rüdiger, and Mandyam V. Srinivasan. "Searching behaviour of desert ants, genus Cataglyphis (Formicidae, Hymenoptera)." *Journal of comparative physiology* 142 (1981): 315-338.

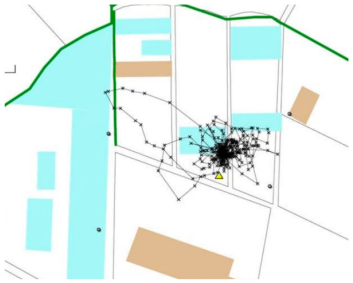


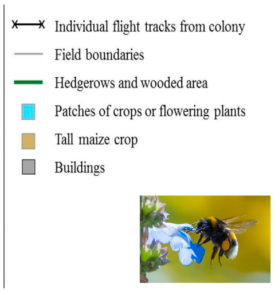
Evidence of using latent spatial learning (unreinforced?) & learning a new path after one experience in rewind task



Clement, Leo, et al.. "Latent learning without map-like representation of space in navigating ants." *bioRxiv* (2024): 2024-08.

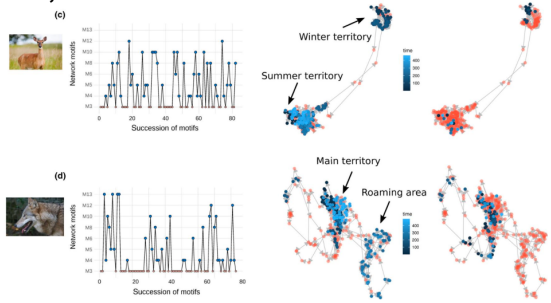
Example 2: Bees make progressively bigger, rotating loops from center hive with experience





Osborne, Juliet L., et al. "The ontogeny of bumblebee flight trajectories: from naïve explorers to experienced foragers." *Plos one* 8.11 (2013): e78681.

Other animals/species: Huge diversity in foraging path patterns, what drives them?



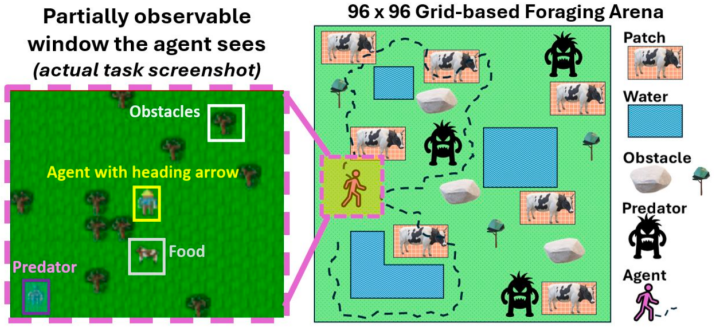
Pasquaretta, Cristian, et al. "Analysis of temporal patterns in animal movement networks." *Methods in Ecology and Evolution* 12.1 (2021): 101-113.

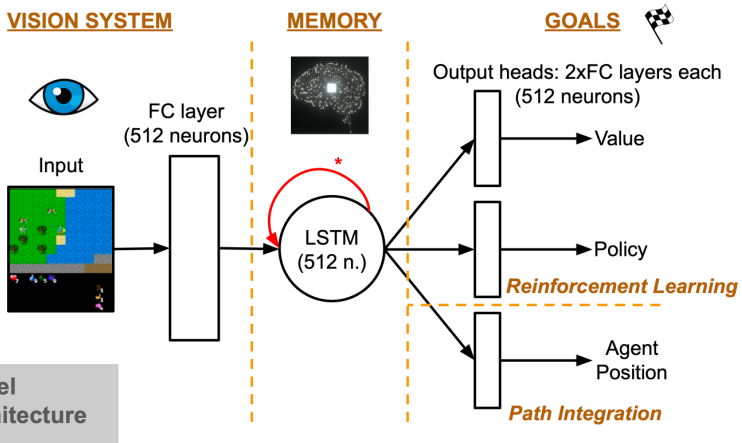
High level questions about exploration and its drivers

- What exactly is **exploration**?
  - In neuroscience, "exploration" is often very overtrained agents in simple binary choice tasks picking a different option than in the previous trial.
  - Can we embed exploration in more interesting conceptual or spatial maps?
- What factors drive exploration in more naturalistic settings?
  - Predators or other dynamic risk pressures perturbing agent paths?
  - Minimizing uncertainty or path integration errors?
  - Exploration being optimal for some reward versus energy or risk tradeoffs?
- Current RL agents in computer sciences are bad at harder exploration tasks, can a better theory of exploration in neuroscience help this bottleneck for both fields?

Matthews, Michael, et al. "Craftax: A Lightning-Fast Benchmark for Open-Ended Reinforcement Learning." *arXiv preprint arXiv:2402.16801* (2024).

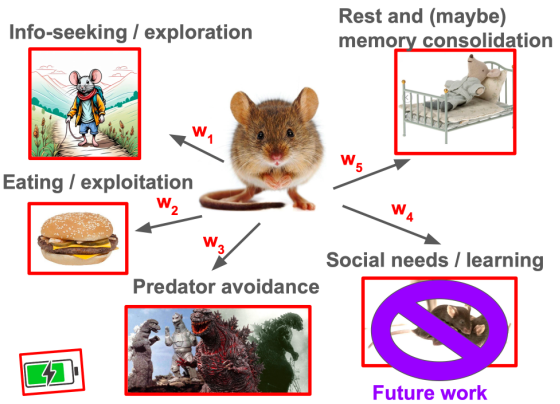
Task Design (new procedurally generated arena each session)



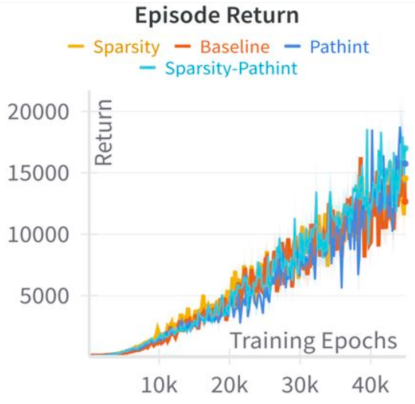


Model Architecture

OPEN-ENDED GOAL: SURVIVE!!



Behavior question:
How does the agent learn to perform well in each sub-goal, and which combination of strategy-switching leads to the longest survival times in the foraging task?



Path integration and biological sparsity?

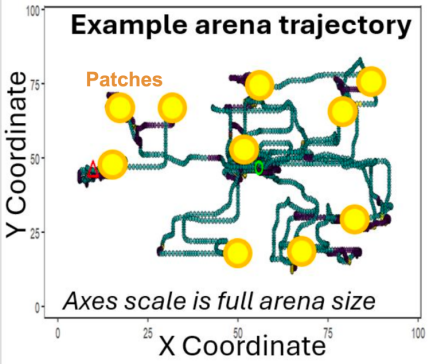
Main conditions tested

	Sparse	Non Sparse
Path Int.	(1, 1)	(1, 0)
No Path Int.	(0, 1)	(0, 0)

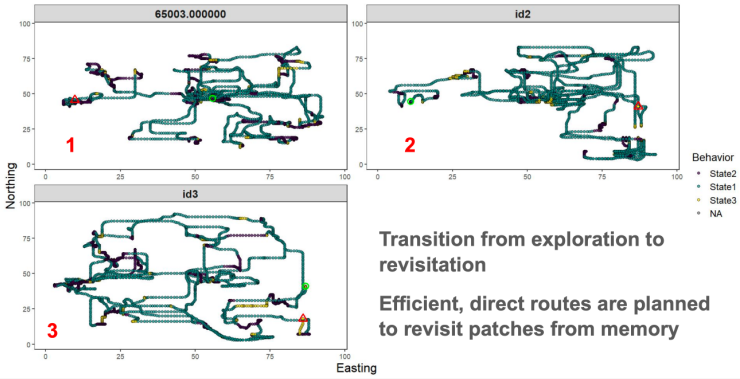
We can train up to 90% sparse networks with no performance loss, connectome-like sparsity [1]

[1] Broido, Anna D., and Aaron Clauset. "Scale-free networks are rare." *Nature communications* 10.1 (2019): 1017.

Agent trajectories look bee-like at the start (sort of)



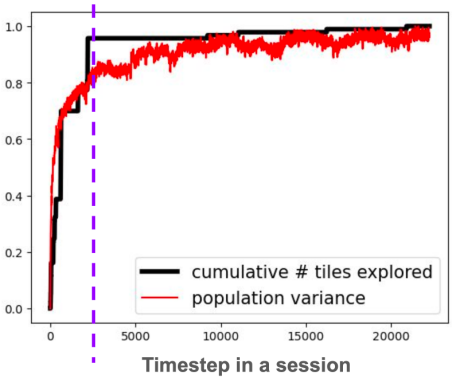
Example paths of one episode split into thirds



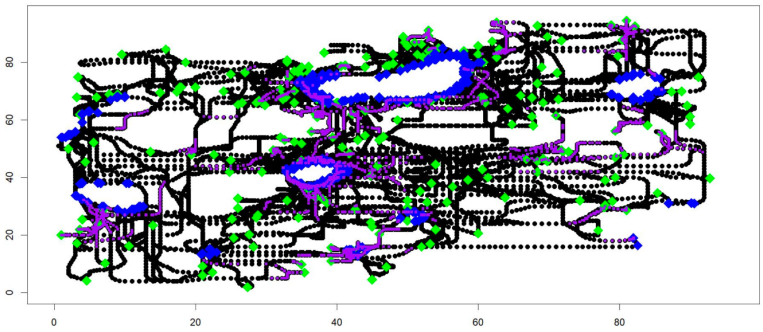
Transition from exploration to revisitation

Efficient, direct routes are planned to revisit patches from memory

Behavior and neural state transition from exploration, to equilibrium revisitation strategy, modulates neural population variance



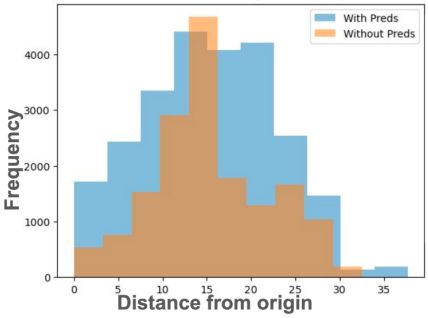
Foraging action-labeled arena example



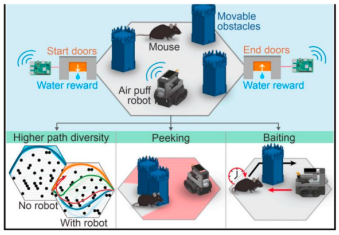
Drink, **eat**, **predator visible**, movement path

Examining exploration drivers

Distance from origin histogram



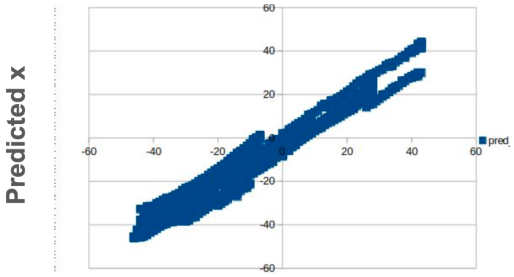
Predators enhance exploration



See *Lai et al, 2024, Cell Reports,* for related lab experiments in mice

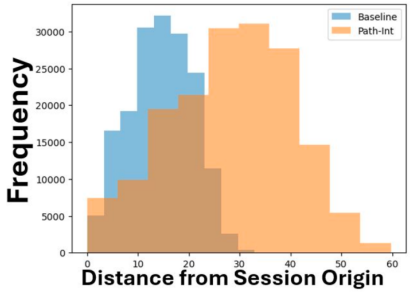
Predicted versus actual distance from origin: path integration works well in our agents.

Gives us a spatial uncertainty readout at every timestep.

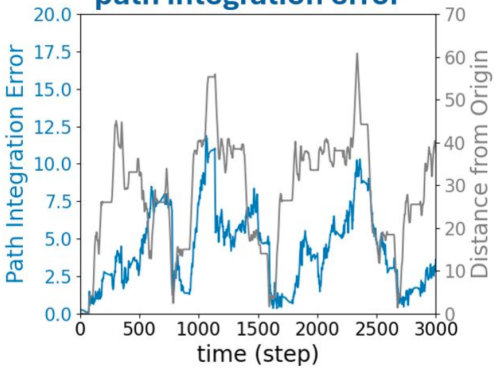


Explicitly path integrating agents explore farther

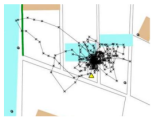
Histogram of distance traveled of agents that path-integrate vs not



Exploration distance vs. path integration error



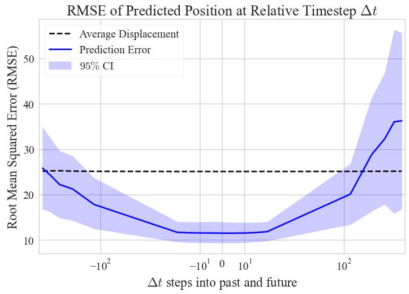
Agents partly return to origin to restore path integration performance



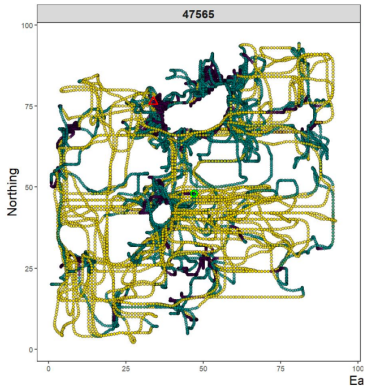
Remember bees and ants!

Decoding analysis of navigational planning and memory:

- Path integration improves decoding performance
- Planning and memory time horizons on order of 100s of time steps

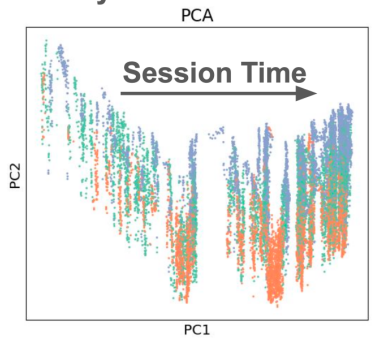


Importantly we find ~10-20 different arena configurations are needed to decode position, rather than time or visual artifacts



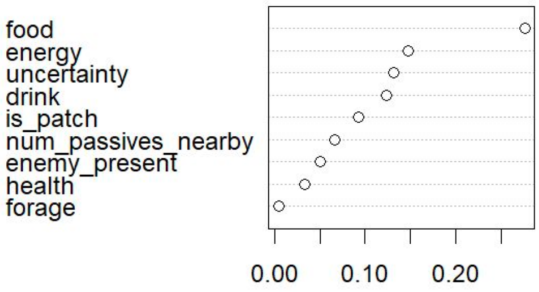
Bayesian path segmentation based on turning angle and step size

Neural manifold analysis of movement states

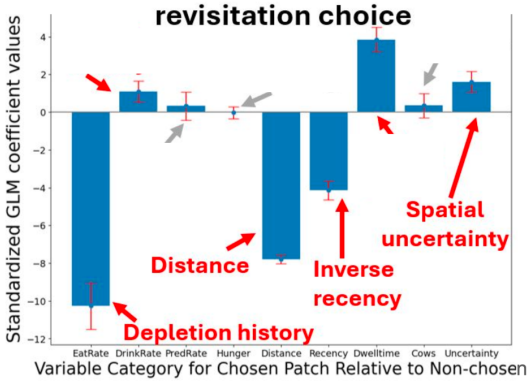


Short, **mid** and **long** range motion cycles with session-level drift

Decision tree importance scoring (not signed) for factors that differentiate states



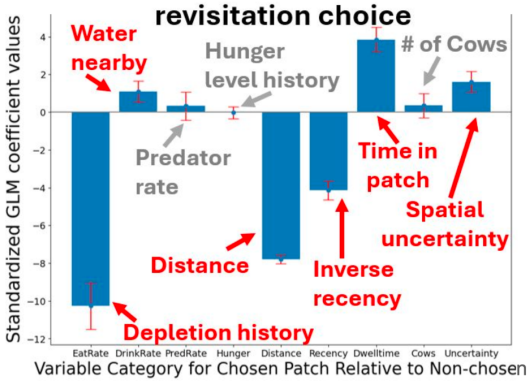
Factors that influence patch revisitation choice



Multi-goal optimization emerges in agents.

One of the unexpected goals is revisiting patches where uncertainty was higher

Factors that influence patch revisitation choice



Multi-goal optimization emerges in agents.

One of the goals is revisiting patches where uncertainty was higher

Conclusions

- Simple agents with orders of magnitudes fewer “neurons” than insects can learn complex world maps without any hard-coded world map architectures.
  - A major challenge for the field is to reconcile neural network-based models with equation-based models.
- Insects have very sophisticated spatial memory that mix features and space. Our models suggest small RL-RNN circuits can support such computations.
- High biologically-relevant sparsity can be achieved in hard tasks without performance loss, makes encoding profiles more brain-like and decodable.

We will continue to do a detailed neuro-style study of which navigational rules DRL agents learn and represent in their activations.

Next steps

- Grid cell augmented RNNs for link to mammalian work
- disRNNs for more interpretable latent states
- Flesh out current results

> Nature. 2018 May;557(7705):429-433. doi: 10.1038/s41586-018-0102-6. Epub 2018 May 9.

Vector-based navigation using grid-like representations in artificial agents

Contradictory Results — Follow this preprint

When and why grid cells appear or not in trained path integrators

Ben Sorscher, Gabriel C. Mel, Aran Nayebi, Lisa Giocomo, Daniel Yamins, Surya Ganguli

doi: https://doi.org/10.1101/2022.11.14.516537

And several other grid cell RNN examples

Other Collaborators

Social Neuroscience (human fMRI)
- Wojciech Zajkowski (NIH)
- Masahiko Haruno (NICT)
- Rei Akaishi (RIKEN)

-Zajkowski, Badman* et al. 2024*

Computational Chemistry
- Zizhang Chen (Brandeis)
- Pengyu Hong (Brandeis)

-Chen, Badman* et al. 2024*

Various ongoing mouse work
- Siyan Zhou (Harvard)
- Christopher Harvey (Harvard)
- Shijia Liu (Harvard)
- Bernardo Sabatini (Harvard)
- Yu Duan (Harvard)

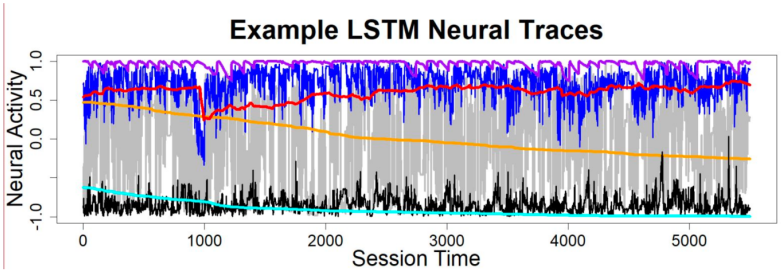
Military AI Ethics & Policy
- Riley Simmons-Edler (Harvard)
- Jean Dong (Harvard, Kennedy School)
- Shayne Longpre (MIT)
- Paul Lushenko (US Army, Cornell)
- Ahmed Mehdi Inane (Mila)

-Simmons-Edler, Badman* et al. 2024*
-multiple policy projects in 2025

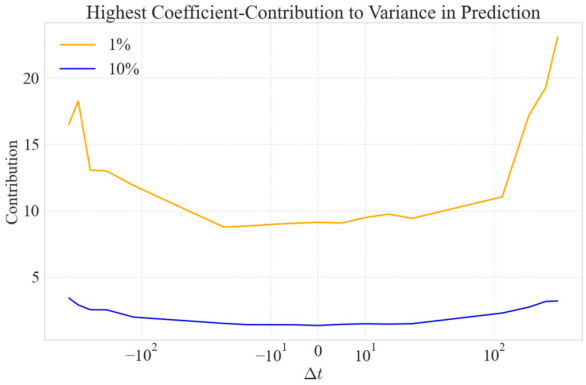
Information Seeking
- Jen Bussell
- Richard Axel
- Larry Abbott
- Ethan Martin-Bromberg
- Christian Marton

-Bussell, Badman et al 2024.

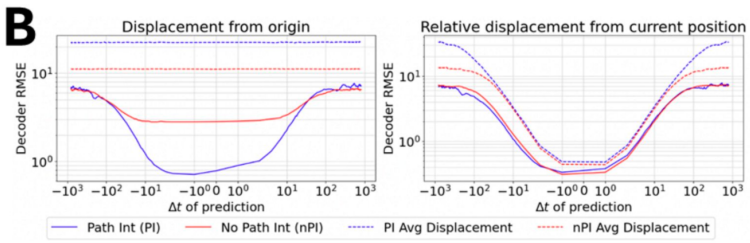
Questions?

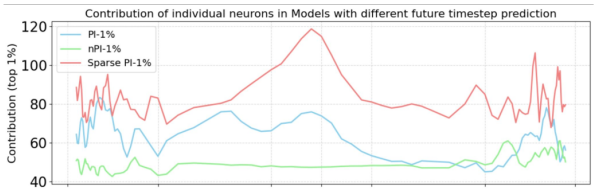


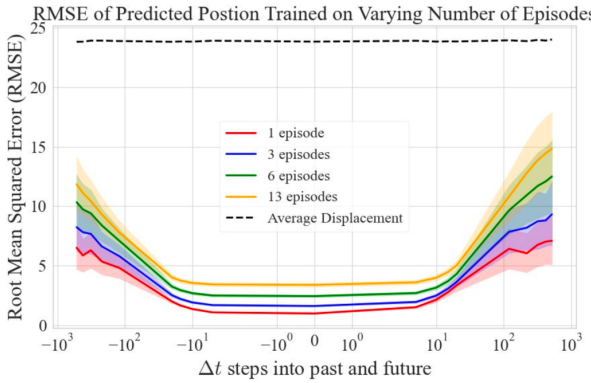
Sparse networks have localized encoding. GLM will explore more soon.



Sparsity and path integration independently improve selective encoding of past/future positions







C Appendix - COSYNE 2025 Abstract

Abstract submitted to the COSYNE 2025 seminar.

ForageWorld: RL agents in complex foraging arenas develop internal maps for navigation and planning

Summary: Foraging, the set of behaviors associated with seeking valuable resources (e.g., food, water) while avoiding danger (e.g., predators), is ubiquitous among organisms, but its neural circuit basis is presently unclear. Most existing work on navigation, a key component of foraging, involves animals in small, fully observable arenas with few-to-no obstacles, partly because recording neural activity in naturalistic settings is challenging. How do animals successfully forage in complex naturalistic environments, especially given realistic neural circuit constraints on capacity and connectivity? To study this question in tractable settings, we designed *ForageWorld*, a procedurally-generated and partially observable arena-based environment, in which artificial agents must satisfy hunger, thirst, and sleep requirements while navigating complex terrain and avoiding predators. We found that agents trained via reinforcement learning (RL) explored the arenas to locate resource-rich patches, and strategically traveled directly between known patches outside the current field-of-view, including ones unobserved for hundreds of timesteps. Moreover, this sophisticated navigational planning was achieved by agents with fewer neurons than insect brains and with sparse connectivity constraints. To analyze these foraging behaviors, we used generalized linear models (GLMs) to quantify how patch features (e.g., distance from agent, historical predator rates, depletion tracking) influence patch revisitation decisions, Bayesian path segmentation to characterize agent behaviors on different timescales, and neural decoding analyses to probe the circuit basis of navigational mapping. Since path integration is thought to be foundational for navigation, we compared agents explicitly trained on path integration to agents that were not. Path-integrating agents explored more of the environment, and in a manner modulated by spatial uncertainty. They also had clearer neural representations of past and future locations, pointing to the emergence of internal maps. Our results pinpoint biologically plausible foraging strategies implemented by neural circuits in navigating organisms with small brains, such as bees and ants.

Additional Detail: Mounting evidence shows that we have underestimated the sophistication with which animals can learn and memorize spatial maps without direct reinforcement [1–3]. Interestingly, deep RL (DRL) neural networks now approach the size and computational sophistication of insect brains [4], but the spatial navigation and planning strategies that DRL agents use in solving exploration-related tasks is almost completely unknown. Deeper analysis into DRL agents can also help infer computational abilities of biological brains with similar size and constraints (e.g. insects).

We built a modular and user-friendly foraging task suite to solve this knowledge gap, where parameters related to patchiness, predator behavior, resource depletion and replenishment rates, etc. can be customized **(Fig. 1)**. Trained agents demonstrate complex insect-like path-finding, path-integration, and strategic path-revisitation behaviors **(Fig. 1, 2)**. Furthermore, path integration as an objective seems to drive exploration in part **(Fig. 1BDE)**, with agents explicitly using path integration exploring farther. Path integration-linked exploration also increases the decodability of neural representations of spatial memory and planning over longer timescales which suggests the emergence of latent cognitive maps **(Fig 2B).** Last, most current neuroAI architectures use highly connected networks that suppress brain-like localized selective encoding and modularity that may emerge from biophysical constraints such as sparsity. Thus, we use connectome-informed sparse networks (90% sparsity) that can nonetheless forage effectively in our task **(Fig 2A)**. Ongoing analysis is now focused on comparing single-neuron encoding profiles of foraging-relevant variables between sparse networks and non-sparse networks, so that our foraging simulations can be honed to have improved bio-relevance to brain data in a larger range of species.

References: [1] L. Clement, S. Schwarz, and A. Wystrach, bioRxiv (2024).; [2] M. Freire, A. Bollig, and M. Knaden, Curr. Biol. **33**, 2802 (2023).; [3] J. H. Wen, … L. M. Giocomo, Nature 1 (2024).; [4] G. C. H. E. de Croon, … J. A. R. Marshall, Sci. Robot. **7**, eabl6334 (2022).; [5] M. Matthews, … J. Foerster, https://arxiv.org/abs/2402.16801v2.; [6] J. Schulman, F., …O. Klimov, arXiv:1707.06347.

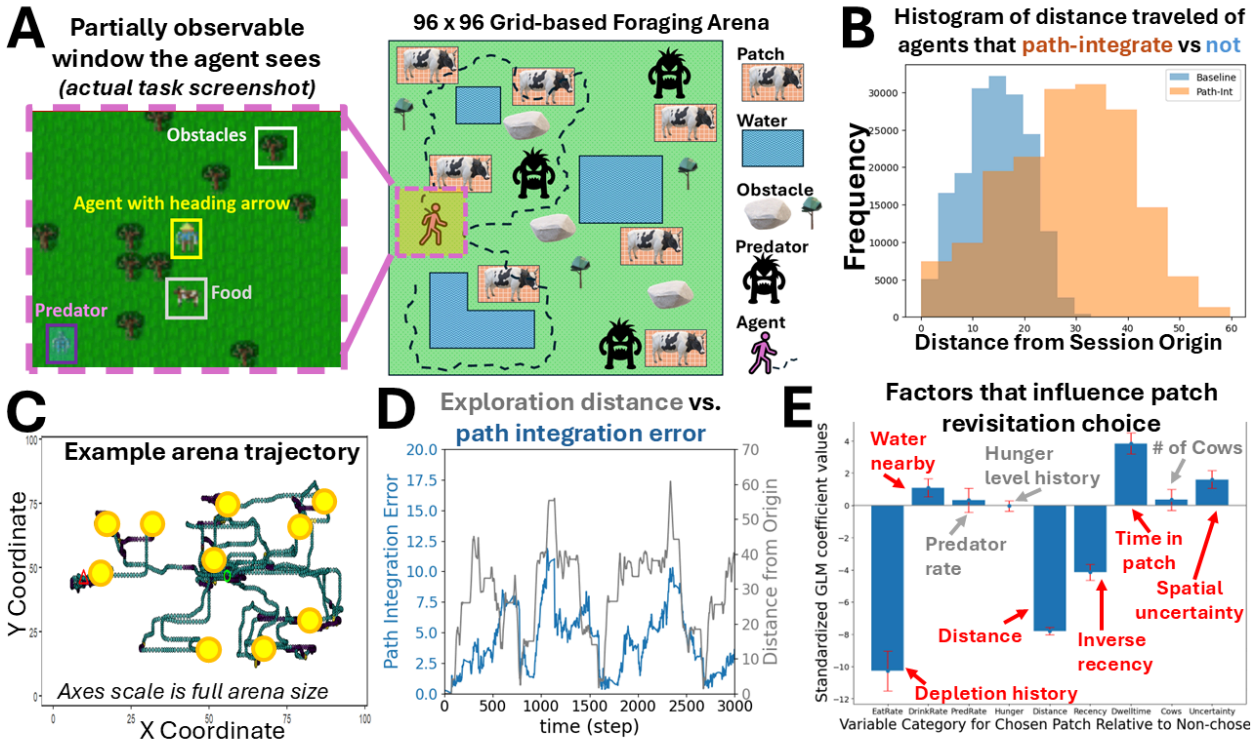


Figure 1, Task design and agent behavior: (A) A schematic of one full foraging environment that is procedurally generated with a new random seed each episode in JAX (right). This novel task suite is substantially modified from the Craftax framework [5] and only requires agents to survive as long as possible by avoiding excessive hunger, thirst, predators, and fatigue. The left insert shows an actual screenshot of what the input task screen "looks" like to an agent with its partial observation window (9 x 7 tiles), which is a fraction of the full arena size (96 x 96 tiles). **(B)** Our DRL agents are trained with proximal policy optimization (PPO) [6] connected to an LSTM (512 neurons) with a feedforward input encoder and output heads (7000 units total) [5]. We train with and without an auxiliary objective to predict current displacement from the episode's origin (the origin is randomly selected for each new episode), in addition to the RL objective. Agents that explicitly learn path integration explore longer distances than agents without. **(C)** Example trajectory of an agent in an example arena, with the origin randomly selected near the center in this trial, and the agent doing explorative loops that return to origin. Annotated segmentation pipelines identify behavioral states such as patch visitation (yellow circles), long range directed motion (green points) and a mixture of area-restricted search and predator avoidance (purple points) **(D)** Plots showing that path integration error worsens with distance from origin in path-integrating agents, but performance is restored after returning to origin. **(E)** Standardized GLM coefficients of patch history variables that an agent uses to decide which patch to revisit in its network of discovered patches. Agents consider multiple significant (red) factors at once, with expected factors such as patch depletion histories and nearness of patches, but also unexpected factors including preferring patches where path integration errors were higher historically (perhaps to reduce uncertainty).

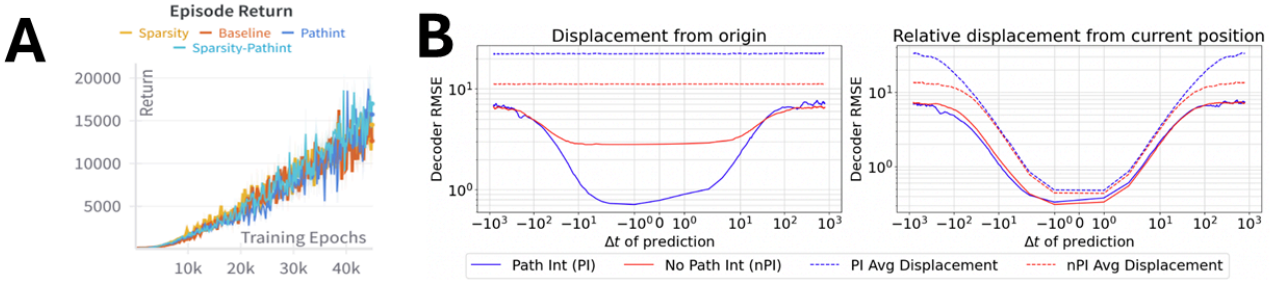


Figure 2, Task learning curve and decoding of navigational variables from recurrent neural network activity: (A) Comparable training performance is shown between agents with 90% sparse versus non-sparse networks, and with and without the path integration objective. **(B)** Solid lines show decoding performance of past and future displacements from the origin (left) and from the agent's current position (right), for agents that have an explicit path integration objective (solid blue) versus those that do not (solid red). Dotted lines show the baseline average displacement within a given time window for comparison, with the path-integrating agent exploring farther. Allocentric representations are clearer (easier to decode) in explicitly path integrating agents (solid blue, left), but results suggest that non-path integrating agents also develop an emergent, but noisier, latent path-integrating ability or "internal map" (solid red, left), Plots are on a log-log scale to show the slower time-scale emergent planning and memorization horizons more clearly (+/- 30 to 40 timesteps).