COMP 400 Final Report: RatSlam 2.0

Ezra Huang* **Jonathan Lamontagne-Kratz*** **Olivier Lefevre***
School of Computer Science
McGill University
{ezra.huang,olivier.lefevre,jonathan.lamontagne-kratz}@mail.mcgill.ca

Isabeau Premont-Schwarz
School of Computer Science
McGill University
isabeau.premont-schwarz@mcgill.ca

* These authors contributed equally. Ordering was determined alphabetically.

Abstract

We introduce RatSLAM 2.0, a novel biologically-plausible SLAM algorithm modernizing the original RatSLAM by integrating a new neocortical-hippocampal-entorhinal model based on the VectorHaSH framework. Our research generalizes the VectorHaSH memory scaffold to three dimensions and non-square modules, confirming the preservation of desirable properties like high capacity scaling linearly with the number of modules. We conducted extensive experiments on hippocampal-sensory learning, comparing Hebbian learning variations with analytic and iterative pseudoinverse methods. Pseudoinverse approaches, particularly the iterative version, demonstrated significantly greater stability for associative memory compared to Hebbian methods which struggled with recall after learning many patterns. We evaluated RatSLAM 2.0 configurations in the AnimalAI environment. The system successfully performed SLAM and tracked pose during continuous motion; however, a significant limitation was its inability to recover from simulated kidnapping events, highlighting a need for future research into robust re-localization strategies.

1 Introduction

The problem of SLAM in robotics has traditionally relied on probabilistic methods, with methods like Kalman filters, particle filters, and GraphSLAM dominating the SLAM landscape [1]. However, SLAM approaches inspired by the brain have also been successful; in 2004, the RatSLAM [2] algorithm successfully implemented a hippocampal model capable of integrating visual and odometric data, with a visual-only variant capable of mapping large areas in real-time [3]. However, despite new discoveries in neuroscience, in particular the discovery of rodent place and grid cells [4] and their use in rodent navigation, little has been done to develop new SLAM algorithms to take advantage of this new neuroscientific inspiration.

More recently, VectorHaSH [5] introduced a neocortical–entorhinal–hippocampal network model that leverages a grid and hippocampal cells scaffold and pseudoinverse learning to achieve high capacity memory. Our RatSLAM 2.0 algorithm heavily leverages and extends the VectorHaSH framework, extending its scaffold from two dimensions to three and extending VectorHaSH's discrete states and simple shifts by exploring shift mechanisms inspired by the original RatSLAM [2].

Preprint. Under review.

1.1 Contributions and Roadmap

In our paper we make the following contributions:

(1) First, we generalize the VectorHaSH memory scaffold to three dimensions and non-square modules, showing that it preserves the nice properties of (1) having large basins and high number of fixed points for coprime lengths across modules, and (2) a required number of hippocampal cells to achieve full capacity that scaling with the number of grid modules and not module sizes.

(2) Second, we extend VectorHaSH's discrete grid module states to accept continuous values and adapt RatSLAM's velocity shift mechanism to work with VectorHaSH's grid modules.

(3) Third, we compare different initialization methods that have been proposed in [5] and [6] and find that they are both sufficient for producing good scaffolds with a maximal number of fixed points requiring the same number of hippocampal cells.

(4) Fourth, we investigate alternatives to analytic pseudoinverse learning used in VectorHaSH to learn hippocampal-sensory associations. Specifically, we benchmark Hebbian learning with some improvements and implement a biologically-plausible pseudoinverse algorithm and compare it against it's analytic counterpart.

(5) Fifth, we test our new VectorHaSH + RatSlam model in AnimalAI [7] environment.

2 Preliminaries

2.1 Extended grid-hippocampal scaffold

We extend the grid-hippocampal scaffold described in VectorHaSH [5] to use 3-dimensional grid modules, so that grid states correspond to a 3-dimensional pose. We also use nonsquare grid modules to allow for varying maximal sizes across dimensions.

Let there be M 3-dimensional grid modules with sizes $\boldsymbol{\lambda} := (\lambda_x, \lambda_y, \lambda_\theta)^{(1)}, \ldots, (\lambda_x, \lambda_y, \lambda_\theta)^{(M)}$ and suppose that $\lambda_d^{(1)}, \ldots, \lambda_d^{(M)}$ are pairwise coprime for all $d \in \{x, y, \theta\}$.

Each grid module has a state $s^{(m)} \in [0,1]^{\lambda_x^{(m)} \times \lambda_y^{(m)} \times \lambda_\theta^{(m)}}$. The grid module state can be compactly represented by a one-hot encoded vector $g^{(m)} \in [0,1]^{\lambda_x^{(m)} \lambda_y^{(m)} \lambda_\theta^{(m)}}$ by flattening.

Since $\lambda_i^{(1)}, \ldots, \lambda_i^{(M)}$ are pairwise coprime for all $i \in \{x, y, \theta\}$, the state of all M grid modules can be compactly represented be the vector:

$$g := \left[g^{(1)} | \ldots | g^{(M)} \right] \tag{1}$$

This vector has length $N_g := \sum_{m=1}^{M} \left(\prod_i \lambda_i^{(m)} \right)$ and there exist $N_{patts} := \prod_{m=1}^{M} \prod_i \lambda_i^{(m)}$ possible fixed points, which we verify experimentally.

2.2 Continuous grid states

In VectorHaSH [5], a continuous extension is proposed, where grid-states are represented as Guassian blobs of activity on a neural sheet. We propose an alternative way of achieving continuous-valued grid states, drawing inspiration from RatSLAM's pose cell network, modeled by a competitive attractor network [2].

Instead of one-hot encoding grid module states to model continuous attractor network (CAN) dynamics, it is possible to apply a smoothing function (such as a softmax) to the state and interpret it as a categorical probability distribution over the finite set of possible states $\Omega^{(m)} = \{(x, y, \theta) \in \{0, \ldots, \lambda_x^{(m)} - 1\} \times \{0, \ldots, \lambda_y^{(m)} - 1\} \times \{0, \ldots, \lambda_\theta^{(m)} - 1\}\}$, representing beliefs of the robot that it's pose corresponds with that grid module state.

This has the benefit that upon receiving small odometry inputs, each grid module state can make precise adjustment to it's belief of pose by moving a small amount of probability mass from one state to the other.

Additionally, the marginal probability distribution $p(x)$, can be recovered by the marginal distributions $p^{(m)}(x)$ for each grid module. Indeed, $p(x)$ can be calculated by $p(x = k) = \prod_{m=1}^{M} p^{(m)}(x = k \mod m)$ for $k \in \{0, \ldots, \prod_{m=1}^{M} \lambda_x^{(m)} - 1\}$. $p(y)$ and $p(\theta)$ can be computed analogously.

2.3 Extended grid-hippocampal initialization methods

While VectorHaSH [5] proposes to initialize the matrix W_{gh} with entries drawn from a standard normal distribution with a given sparsity, we also test an alternative method of initialization where we initialize W_{gh} with entries drawn the a distribution with nonzero mean and standard deviation used in [6]. We test both methods and find both of them suitable for ensuring sparsity in hippocampal activation without sacrifices in scaffold properties. The mathematics to calculate parameters to achieve certain sparsity ψ for both methods can be found in Appendix 5.

2.4 Grid-hippocampal scaffold velocity shift methods for SLAM

In VectorHaSH [5], the velocity shift mechanism does a simple translation of activity in each grid module. However, for SLAM, we wish to be able to capture small incremental movements in short time intervals in response to odometer inputs. Therefore we take inspiration from RatSLAM's ratshift operation [2], where activity becomes spread out across cells when recieving input.

Ratshift We test the method described in RatSLAM [2] to inject activity into each grid module. Given an input speed v, direction θ, and angular velocity ω, where k_x, k_y, k_θ are conversion constants from real world values to grid cells, the scaled components of v for the x and y dimensions as well as for ω are calculated:

$$v_x = v\cos(\theta)k_x, \quad v_y = v\sin(\theta)k_y, \quad v_\theta = \omega k_\theta$$

Each component is decomposed into integer and fractional parts:

$$\delta_x, \delta_{fx} = \lfloor v_x \rfloor, v_x - \lfloor v_x \rfloor \quad \delta_y, \delta_{fy} = \lfloor v_y \rfloor v_y - \lfloor v_y \rfloor, \quad \delta_\theta, \delta_{f\theta} = \lfloor v_\theta \rfloor, v_\theta - \lfloor v_\theta \rfloor$$

Next the alpha weight tensor, $\alpha \in \mathbb{R}^{2\times2\times2}$, is calculated by

$$(\alpha)_{ijk} = g(\delta_{fx}, i) \cdot g(\delta_{fy}, j) \cdot g(\delta_{f\theta}, k) \qquad (i, j, k \in \{0, 1\})$$

where

$$g(a, b) = \begin{cases} 1 - a & \text{if } b = 0 \\ a & \text{otherwise} \end{cases}$$

Finally, activity in each grid module is updated by (superscripts ommited for brevity):

$$s_{nlm} = \sum_{i,j,k=0}^{1} \alpha_{ijk} s_{i',j',k'}$$

where i', j', k' are given by:

$$i' = (n + i - \delta_x) \mod \lambda_x \quad j' = (l + l - \delta_y) \mod \lambda_y \quad k' = (m + i - \delta_\theta) \mod \lambda_\theta$$

2.5 Smoothing methods for SLAM

If probability mass in each module continues to spread out in each module (as a result of odometry shifts), the robot will be left with a uniform distribution about it's pose. This contrasts with the property of the RatSLAM algorithm that, given enough time without moving, the robot will converge on a belief about it's pose [2]. Therefore, it is necessary to use a smoothing function after each velocity shift that will ensure that (1) activity in each module remains a probability distribution and that (2) the distribution will not tend towards a uniform distribution when receiving continuous no or small input.

Argmax smoothing For the sake of comprehensiveness, argmax smoothing is the same function defined in VectorHaSH [5]. It has the drawback that it is imprecise. Specifically, it will lose information when odometry inputs are small or cause shifts that distribute probability mass equality between two grid cells, leading to accumulating odometry errors.

Formally, it can be expressed in the following way (the grid module superscript is dropped for brevity)

$$i, j, k = \arg\max_{ijk}(s) \tag{2}$$

$$s'_{x,y,\theta} = \delta_{ix}\delta_{jy}\delta_{k\theta} \tag{3}$$

, where δ_{ij} is the Kronecker delta, $\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$

Softmax smoothing Softmax smoothing is commonly used in neural networks as an activation function. Here, τ denotes temperature.

$$\sigma(x)_{ijk} = \frac{e^{x_{ijk}/\tau}}{\sum_{xy\theta} e^{x_{ijk}/\tau}} s' = \sigma(s) \tag{4}$$

Polynomial smoothing This type of smoothing is similar to softmax smoothing, but has the benefit that it will not add probability mass to cells that did not initially have any, leading to decreased spreading of probability mass to far-away locations. κ is a hyperparameter.

$$\phi(x)_{ijk} = \frac{x^{\kappa}_{ijk}}{\sum_{xy\theta} x^{\kappa}_{xy\theta}} \tag{5}$$

RatSLAM smoothing When using Ratshift, we also use the CAN dynamics proposed in RatSLAM [2], which happen in the following four steps. Once again, superscripts are ommited for brevity.

1. Spatial Update. Let $\epsilon \in \mathbb{R}^{\lambda_x \times \lambda_y}$ be a 2D Gaussian kernel with radius r centred at $(0, 0)$. For each angle θ, the module state is updated by

   $$s_{xy\theta} = s_{xy\theta} + \sum_{i=-r}^{r} \sum_{j=-r}^{r} \epsilon_{ij} \cdot s_{x+i,y+j,\theta}$$

   where $r = \left\lfloor \frac{\lambda_x}{2} \right\rfloor$ (assuming $\lambda_x = \lambda_y$ with circular padding added as needed).
2. Orientation Update. Let $\delta \in \mathbb{R}^{2\gamma+1}$ be a 1D Gaussian kernel applied over the orientation dimension θ. This kernel only affects the γ-neighborhood around each orientation layer. The orientation update is:

   $$s_{xy\theta} = s_{xy\theta} + \sum_{k=-\gamma}^{\gamma} \delta_k \cdot s_{x,y,\theta+k}$$

   with circular indexing on θ.
3. Global Inhibition. Global inhibition subtracts from all cells based on the global maximum:

   $$s_{xy\theta} = \max\left(0, s_{xy\theta} + \lambda \cdot (s_{xy\theta} - \max_{x',y',\theta'} s_{x'y'\theta'})\right)$$

4. Normalization. Finally, the updated grid cell matrix is normalized:

   $$s_{xy\theta} = \frac{s_{xy\theta}}{\sum_{x',y',\theta'} s_{x'y'\theta'}}$$

2.6 Hippocampal-sensory learning

In this section we describe a collection of methods we tested for learning associations between hippocampal and sensory cells. For this section, we denote the i-th sensory input by s_i, the associated hippocampals state by h_i, the set of all seen hippocampal states by H and the set of all seen sensory inputs by S. The goal is to continuity update the matrices W_{hs} and W_{sh} upon receiving a new input (s^j, h^j) to minimize the errors $\mathcal{L}(S, W_{sh}H)$ and $\mathcal{L}(H, W_{hs}S)$. For SLAM, it is essential for neurons to maintain both plasticity and stability; otherwise, it may be the case that the robot is unable to map out new locations or unable to recall past locations.

Hebbian learning The simplest learning algorithm is Hebbian learning. Upon receiving a new pair (s^j, h^j), we compute the matrix updates:

$U_{sh} = \frac{s^j (h^j)^T}{||h^j||}$

$U_{hs} = \frac{h^j (s^j)^T}{||s^j||}$

and update the weight matrices by:

$W_{sh} = W_{sh} + U_{sh}$

$W_{hs} = W_{hs} + U_{hs}$

Hebbian learning with zero-centering One problem with the above naive method is that in VectorHaSH, each component of hippocampal input vectors is positive. More precisely, we have that $h_i \sim \phi(X - \Theta)$ where $X \sim \mathcal{N}(\mu, \sigma^2)$ and it is not necessarily the case that $\mu = 0$. The problem arises when sensory inputs are recovered after learning many many sensory-hippocampal associations. Indeed, let s^* be the recovered sensory input resulting from the assumed-correct hippocampal vector h^ν.

$$s_i^* = \sum_{j=1}^{N_h} (W_{hs})_{ij} h_j^v \tag{6}$$

By the definition of W_{hs},

$$s_i^* = \sum_{j=1}^{N_h} \sum_{\mu=1}^{N_{patts}} \frac{s_i^\mu h_j^\mu h_j^v}{||h^\mu||^2} = s_i^v \sum_{j=1}^{N_h} \frac{h_j^v h_j^v}{||h^v||^2} + \sum_{\mu \neq v}^{N_{patts}} \sum_{j=1}^{N_h} \frac{s_i^\mu h_j^\mu h_j^v}{||h^\mu||^2} \tag{7}$$

Hence,

$$s_i^* = s_i^v + \sum_{\mu \neq v}^{N_{patts}} \sum_{j=1}^{N_h} \frac{s_i^\mu h_j^\mu h_j^v}{||h^\mu||^2} \tag{8}$$

If h_j^μ is zero-centred, since h is a high-dimensional vector, it is almost certain that h^μ and h^ν are orthogonal. However, it would be incorrect to assume this since h_j is the result of a rectification function. Therefore, it is necessary to subtract the expectation of h_j from each variable. Let $\bar{h} = E[h] = [E[h_j], \ldots, E[h_j]]$. The derivation to calculate $\bar{h}$ can be seen in Appendix 5.

The learning rules are hence updated to be:

$U_{sh} = \frac{s^j (h^j - \bar{h})^T}{||h^j||}$

$U_{hs} = \frac{(h^j - \bar{h})(s^j)^T}{||s^j||}$

Moreover, the equations for calculating s are modified to account for this translation:

$h - \bar{h} = W_{hs} s$

$s = W_{sh}(h - \bar{h})$

Hebbian learning with past input reinforcement One approach to improve stability without compromising plasticity is to combine new and previous inputs when learning a new input. This way, when new images are learned, you learn based off the difference in output instead of solely the raw input. Mathematically, this can be expressed as:

$U_{sh} = \frac{s^j (h^j - W_{hs} s^j)^T}{||h^j||}$

$U_{hs} = \frac{(h^j)(s^j - W_{sh} h^j)^T}{||s^j||}$

Then the rest remains the same as in Hebbian learning.

Analytic pseudoinverse learning For analytic pseudoinverse learning, we can take advantage of methods like the Moore-Penrose pseudoinverse algorithm like what was used in VectorHaSH [5]. In this method, after receiving a new pair (s^j, h^j) the matrices W_{hs} and W_{sh} are recomputed. Note

that for this method, H is a fixed matrix of all possible hippocampal fixed points, not just the ones currently seen, and S is a matrix filled with 0s, with nonzero columns present for each sensory input seen.

$W_{hs} = HS^+$

$W_{sh} = SH^+$

Iterative pseudoinverse learning .

Another biologically-plausable way of learning these two matrices learned to calculate the analytic pseduoinverse [8] [9]. For our experiments, we implement the stationary version of the algorithm described in [8] with hyperparameters that can be found in 1. For full algorithmic details, see Algorithm 1

Algorithm 1: Bidirectional iterative pseudoinverse learning

Input: S, H

Output: Updated weight matrices W_{hs}, W_{sh}

LEARNED_PSEUDO_INVERSE_HS(INPUT, OUTPUT)

  Compute $b_k = (\mathbf{\Theta}_{hs} \cdot \text{input})/(1 + \text{input}^T \mathbf{\Theta}_{hs} \text{input})$

  Update inhibition matrix:

  $\mathbf{\Theta}_{hs} = \mathbf{\Theta}_{hs} - \mathbf{\Theta}_{hs} \cdot \text{input} \cdot b_k^T$

  Update weight matrix:

  $W_{hs} = W_{hs} + (\text{output} - W_{hs} \cdot \text{input}) \cdot b_k^T$

LEARNED_PSEUDO_INVERSE_SH(INPUT, OUTPUT)

  Similar procedure as $learned\_pseudo\_inverse\_hs$ applied to W_{sh}

2.7 RatSLAM 2.0 algorithm

In this section we describe different variations of the RatSLAM 2.0 algorithm.

We denote storing strength, either hard or soft, to be what hippocampal state we store our current sensory state. A hard store, would be to argmax smooth our modules before storing our sensory state. Formally we would do (2), then perform hippocampal-sensory learning with the hippocampal state obtained from our argmax smoothing.

A soft store, would be to not do this, we simply do hippocampal-sensory learning with our current module states.

We will additionally test two different ways of merging grid states obtained from sensory cells and those from the current module states. We present additive merging, given two flattened grid states as in (1):

$$\text{Given } g_1 := [g_1^{(1)}|...|g_1^{(M)}], g_2 := [g_2^{(1)}|...|g_2^{(M)}] \text{ we return } g = g_1 + g_2$$

Similarly, we present multiplicative merging :

$$\text{Given } g_1 := [g_1^{(1)}|...|g_1^{(M)}], g_2 := [g_2^{(1)}|...|g_2^{(M)}] \text{ we return } g = g_1 \odot g_2$$

Where $\odot$ represents the element-wise product. We present the main algorithm in (2). We also see (5), where we have the alternative hard and soft storing methods.

We also compute certainty of the position beliefs. For this we use the p-th moment of a circular distribution. The p-th moment of a circular distribution is defined [10]:

$$m_p = \frac{1}{n} \sum_{k=1}^{n} (e^{ix_k})^p = \frac{1}{n} \sum_{k=1}^{n} \cos(px_k) + i \sin(px_k) \tag{9}$$

For each grid module, we can calculate the mean μ along each dimension by $\mu = E[X] = Arg(m_1)$.

Then by definition,

$$P(|X-\mu|>k) \leq \sum_{i \notin \{\lceil \mu-k \rceil, \ldots, \lfloor \mu+k \rfloor\}} P(X=i) \leq p \tag{10}$$

This gives us a way to calculate the certainty of the current grid state. We can use this to determine if we should store a new memory or not. We can also use this to determine if we should use a hard or soft store.

Using this we also define variations of the algorithm where we store sensory inputs always, or only when our central position belief has changed. Formally, using the 1st moment of the circluar distribution, as above. Along each dimension, if the mean has changed by some heuristics $limit_x, limit_y, limit_\theta$, then we store a new memory.

Algorithm 2: Main Loop

$s \leftarrow$INITIALIZE MODULES($\boldsymbol{\lambda}$)
$g \leftarrow$ obtain g by flattening according to equation 1
$W_{hg} \leftarrow$INITIALIZE $W_{hg}(\mu, \sigma, \gamma)$
$W_{gh} \leftarrow$ Calculate W_{gh} according to equation 3
while $true$ **do**
  take action u_t
  $dx, dy, d\theta \leftarrow$ GET ODOMETRY()
  $g' :=$RATSHIFT($g, dx, dy, d\theta$)
  $s \leftarrow$ GET SENSORY()
  $certainty \leftarrow$ ESIMATE CERTAINTY(g', k) 9
  $\bar{h} \leftarrow$HIPPOCAMPAL FROM SENSORY(s)
  $\bar{g} \leftarrow$GRID FROM HIPPOCAMPAL(h)
  if *always store* **then**
    if $certainty \geq p_{threshold}$ **then**
      $g \leftarrow$ADDITIVE MERGE(g', $\bar{g}$) or MULTIPLICATIVE MERGE(g', $\bar{g}$)
      STORE MEMORY(g, s)
    else
      $g \leftarrow$ADDITIVE MERGE(g', g) or MULTIPLICATIVE MERGE(g', g)
      STORE MEMORY($\bar{g}$, s)
    end if
  else
    $n \leftarrow$ [MEAN BELIEF(g) - MEAN BELIEF(g') > limits]
    if *n* **then**
      STORE MEMORY(g, s)
    $g \leftarrow$ADDITIVE MERGE(g', g) or MULTIPLICATIVE MERGE(g', g)
  end if
end while
MEAN BELIEF(g)
  return estimate first moment of g using 9

3 Experiments

3.1 Testing of grid-hippocampal initialization methods

To verify that the two different initialization methods are equivalent, we run experiments to ensure that the average hippocampal neuron activation error when a hippocampal fixed state is projected up to grid cells, denoised, and projected back down the hippocampal layer is independent of initialization method. For this experiment, a subset of varying size of all possible grid states is selected. The scaffold then attempts to learn all possible patterns, with the error measured after each run.

We note that the way the hippocampal neuron error changes in response to differences in the total number of hippocampal cells and number of patterns to memorize identically. Importantly, it preserves

the property of VectorHaSH that the error scales exponentially with the number of hippocampal cells, hence. See Figure 1.

3.2 Properties of extended grid-hippocampal scaffold

We also note that when extending the scaffold to 3 dimensions, the error of recovered hippocampal cells stay the same. Indeed, in Figure 2 and Figure 3, we observe that regardless of the dimensionality of the modules, the error remains the same, and moreover, it requires the same amount of hippocampal cells to obtain optimal performance (learns all patterns). Another important result of VectorHaSH that we verify is that the number of hippocampal cells needed scales linearly with the number of modules, and not the total capacity of the scaffold [5]. This result is preserved in the extended scaffold, which can be seen in Figure 4 and 5.

3.3 Testing of iterative pseudoinverse hyperparameters and test against analytic version

To verify that the iterative pseudoinverse algorithm achieves good performance, we compare it against an analytic solution. Interestingly, the iterative solution achieves lower error when recovering images for the entire dataset, both up to and past the theoretical limit of perfect performance as seen in Figure 6. Moreover, from these figures we see that the analytic solution fails catastrophically sooner than the iterative solution, caused by non-convergence of the algorithm used in PyTorch used to compute the pseudoinverse.

3.4 Testing of Hebbian learning variations

We test different variations of Hebbian learning on a whitened MNIST dataset. First, a grid-hippocampal scaffold is generated with hyperparameters that can be found in Table 2. Then, images from MNIST are learned sequentially. After each image is learned, three metrics are calculated: (1) the average L2 error between recovered and original images across the dataset of all learned images, (2) the average L1 error across the difference of sensory neuron activation and the true image is calculated to measure stability, and (3) the average L1 error across the difference of sensory neuron activation and the true image is calculated to measure plasticity. As a baseline, we include errors that are determined by using the mean of the dataset as a recovered image. To estimate uncertainty, we average results over 10 runs with randomly selected MNIST images and a random scaffold, with coloured regions indicating 1 standard deviation away from the mean.

A noticeable result is that none of the Hebbian learning methods have sufficient stability to recall the first image better than a mean of the dataset after 100 images are learned, as seen in Figure 10. This stands in contrast to the analytic and iterative pseudo-inverse methods, which are able to recall the image even 400 images are learned, visible in Figure 9.

Another result is that using the mean-centring fix improves the stability and overall performance across the dataset while impeding plasticity, as seen in Figures 8, 10, and 12.

Another result is that using past input reinforcement is critical to achieving performance. Without it, the weights of the projection matrices tend to explode, leading to massive incorrectly-scaled outputs, visible in Figure 7.

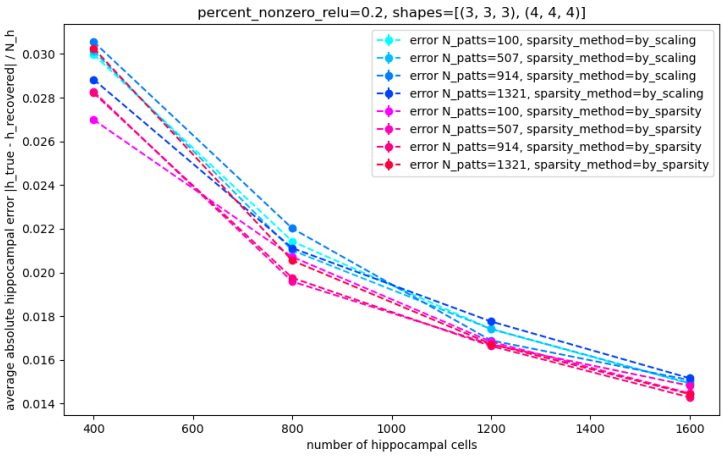

Figure 1: Change of average hippocampal cell neuron error in response to varying number of total hippocampal cells and total number of patterns stored. Notice that both methods are essentially equivalent with marginal (± 0.001) differences, as both methods produce exponential decay curves in error with respect to number of hippocampal cells, unaffected by the number of patterns stored when N_{patts} is less than the theoretical capacity, 1728.

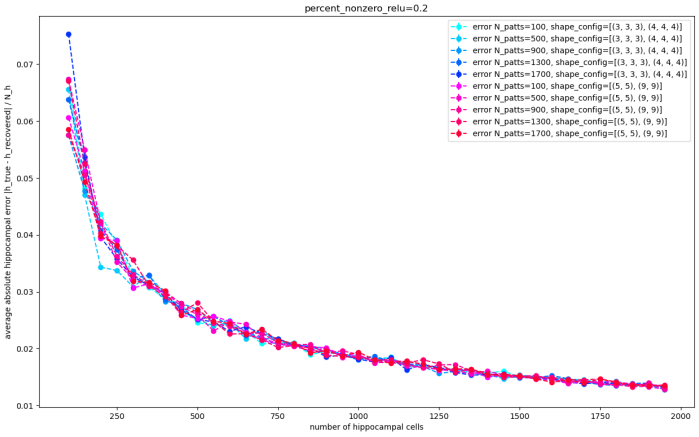

Figure 2: Average absolute hippocampal cell error as a function of number of hippocampal cells for two and three dimensional grid modules. Notice that how that regardless of the configuration of grid module shapes, the exponential decay of the error with respect to the number of hippocampal cells remains the same, with N_{patts} not influencing performance as long as it remains less than the theoretical capacities of 1728 and 2025.

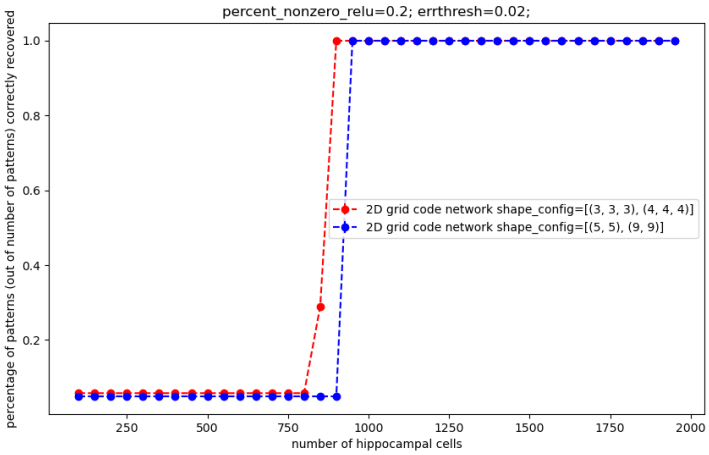

Figure 3: Number of hippocampal cells needed to achieve full capacity for two and three dimensional grid modules. We remark that regardless of shape, the number of hippocampal cells required to achieve full capacity is independent of grid module shape configuration. In this case, the percentage of patterns correctly recovered is defined as the percentage of all grid states in which the L1 error between an original grid state and projections from the grid to hippocampal layer, back to a recovered grid is less than 0.02, i.e. $||g_{\vec{x}} - W_{gh}(\text{ReLU}(W_{hg}g_{\vec{x}}))||_1 < 0.02$ for $\vec{x}$ a one-hot encoded state.

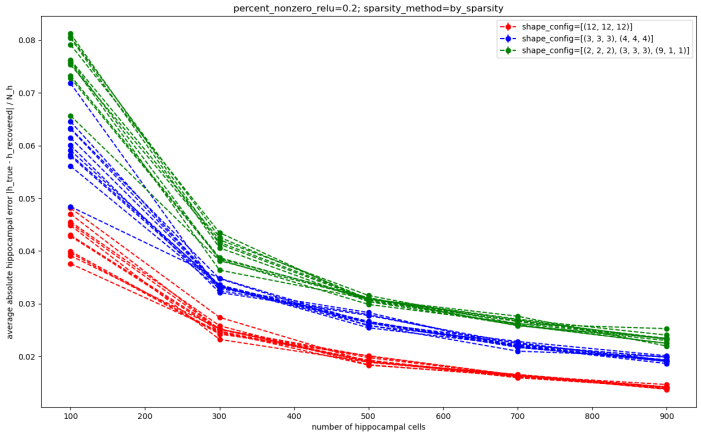

Figure 4: Average absolute hippocampal cell error as a function of number of hippocampal cells for varying shapes sizes. Notice that like VectorHaSH, the number of hippocampal cells required to achieve error between a certain threshold scales not with the size of grid modules, but the number of grid modules. The multiple lines for each shape configuration correspond to a different numbers of total patterns tested, all of which are under the theoretical capacity of each shape configuration.

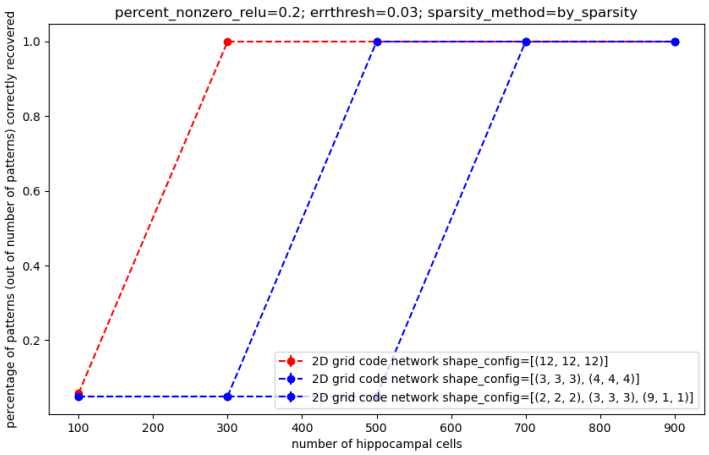

Figure 5: Number of hippocampal cells needed to achieve full capacity for a varying number of grid modules. We note that this critical number of hippocampal cells scales not with the sizes of grid modules, but with the total number of grid modules. In this case, the percentage of patterns correctly recovered is determined like in Figure 3 but with a threshold of 0.03 instead of 0.02.

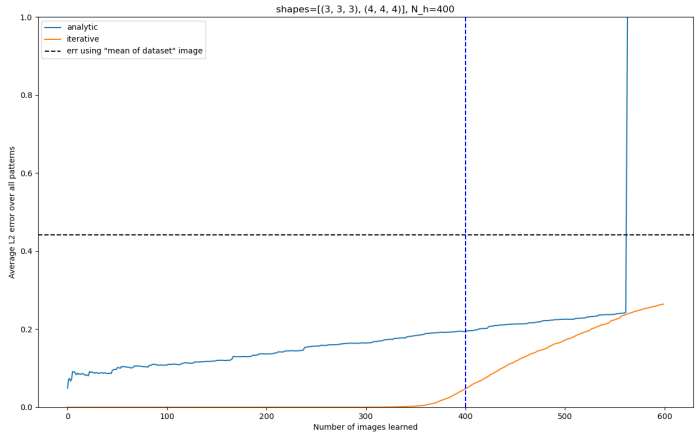

Figure 6: Average L2 error per neuron when recovering MNIST patterns for analytic and pseudoinverse methods. Like in VectorHaSH, we see that the error across the dataset remains low until the number of images stored reaches the number of hippocampal cells, after which the error begins to rapidly increase and recalled images gradually deteriorate towards a mean of all images learned. We also observe that the iterative pseudoinverse solution consistently outperforms the analytic pseudoinverse, and moreover, at a number of images around 560, the analytic pseudoinverse catastrophically fails while the iterative pseudoinverse continues to gradually decrease in performance.

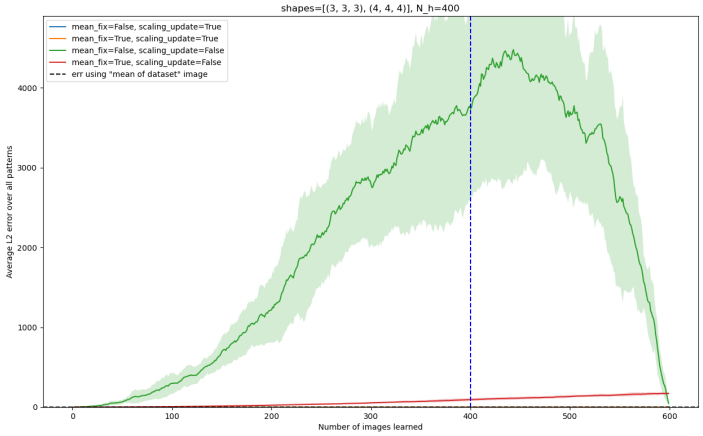

Figure 7: Average L2 error per neuron between recovered and original images across all learned images as more images are learned for different variations of Hebbian learning. It is evident that the error explodes in comparison with pseudoinverse methods, with errors orders of magnitude worse than a mean of the dataset. Still, we observe that naive Hebbian learning, without zero-centering (mean_fix) and past input reinforcement (scaling_updates), performs considerably worse than all other methods.

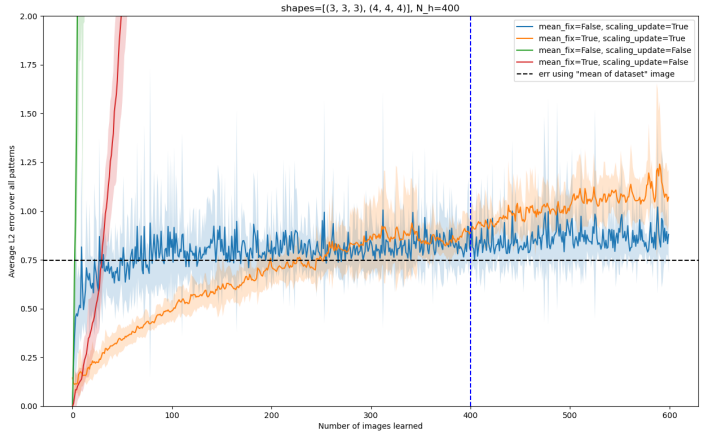

Figure 8: Average L2 error per neuron between recovered and original images across all learned images as more images are learned for different variations of Hebbian learning, zoomed in on the y axis. It is evident from the differences in orange and blue curves that using the zero-centring fix (mean_fix) slows down deterioration of performance towards a mean of the dataset. It is also evident from the differences between the blue/orange curves and red/green curves that past input reinforcement (scaling_update) is essential to avoid massive errors from accumulating.

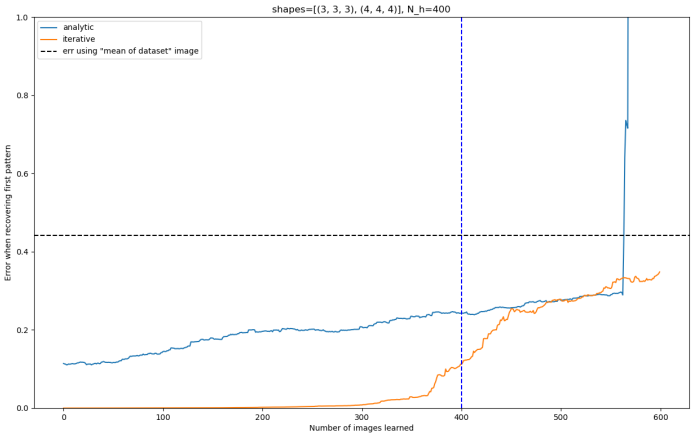

Figure 9: Average L1 error of the recovered first image as more images are learned for analytic and pseudoinverse learning.

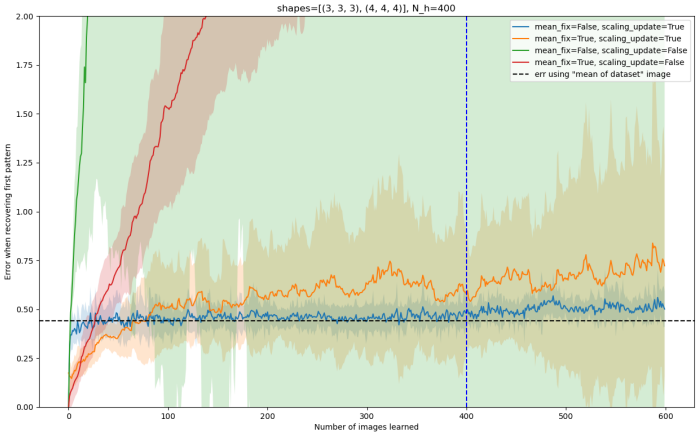

Figure 10: Average L1 error of the recovered first image as more images are learned for different variations of Hebbian learning, zoomed in on the y axis. Note that in contrast to pseudoinverse methods, performance is relatively bad, degrading to a mean of the dataset after only about 28 images are learned whereas pseudoinverse methods are able to maintain good performance up to and beyond 400 images.

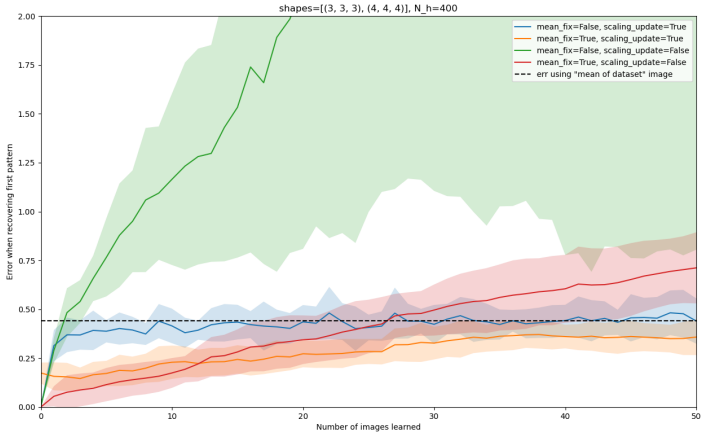

Figure 11: Average L1 error of the recovered first image as more images are learned for different variations of Hebbian learning, zoomed in on the x and y axes. Notice how using the zero-centring fix (red and orange curves) is essential to slow down the degradation of performance towards a mean of the dataset, in comparison to methods not using it (green and blue curves).

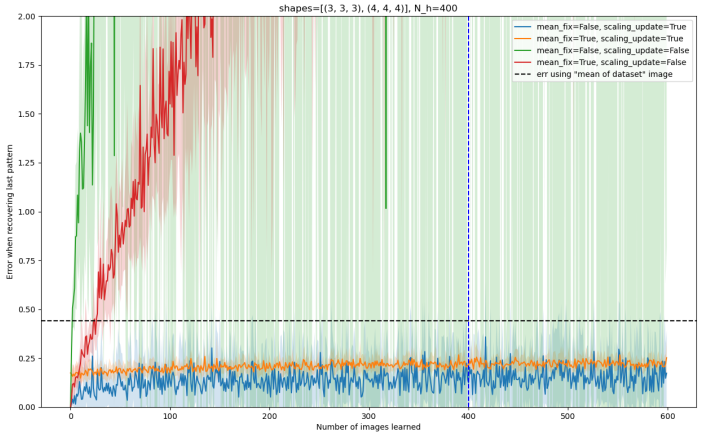

Figure 12: Average L1 error of the last-learned image as more images are learned for different variations of Hebbian learning, zoomed in on the y axis. We notice that when using past reinforcement, the mean-centering fix (orange curve) does on average worse than not using it (blue curve), indicating decreased plasticity. However, both methods result in acceptable performance.

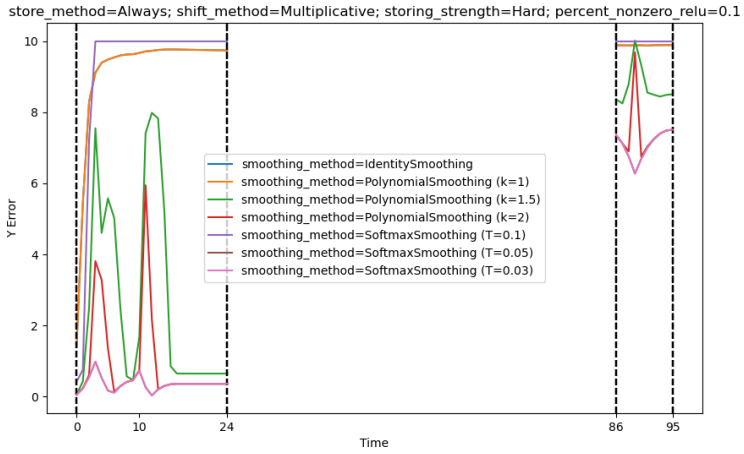

Figure 13: y dimension error for the AnimalAI test of the RatSLAM algorithm used with various smoothing methods applied after the the Ratshift operation. We notice that softmax smoothing with a low temperature performes best (pink curve). Overall, smoothing methods that caused greater inhibition performed better (higher k values for polynomial smoother and lower T values for softmax smoothing).

3.5 Testing of the RatSLAM 2.0 algorithm

In this section, we test out the RatSLAM 2.0 algorithm described above. Specifically, we test out various configurations of the algorithm, including of hard vs. soft storing, storing only when in new grid states vs. always, multiplicative vs. additive shifts, and different smoothing methods when applied on top of the RatSLAM-style CAN dynamics and global inhibition. A full set of other hyperparameters used to initialize the scaffold and hippocampal-sensory layer in these tests is available in the appendix in Table 3.

In this experiment, the agent moves for ten time steps forward and turns 90 degrees to the right in increments of six degrees (15 time steps). Then, it is moved back to it's original position without being given any observations or odometric data, simulating the effect of being kidnapped. Finally, it moves forward for ten timesteps again, testing if the agent is able to recover from being kidnapped.

In all cases, we find that the agent is not able to recover from being kidnapped. However, in the first 24 timesteps, the first ten of which are spent moving forward and the next 15 of which are spent turning right, we find that softmax smoothing with a multiplicative shift is best with the lowest error, which is evident in Figures 13 and 14. We also see that polynomial smoothing does better with multiplicative smoothing than additive smoothing, which is evident in Figures 15 and Figures 16. In all figures, results from the y and θ dimensions were shown as they were most illustrative, with similar but less pronounced effects visible in the x dimension as well. Finally, we see that softmax smoothing is not affected by multiplicative versus additive smoothing, which is illustrated in Figures 17 and 18

In terms of error in estimated pose, we found no differences between hard vs. soft smoothing and percent nonzero relu, which makes sense because as long as the scaffold maintains it's fixed-point properties, these would only affect hippocampal-sensory interactions.

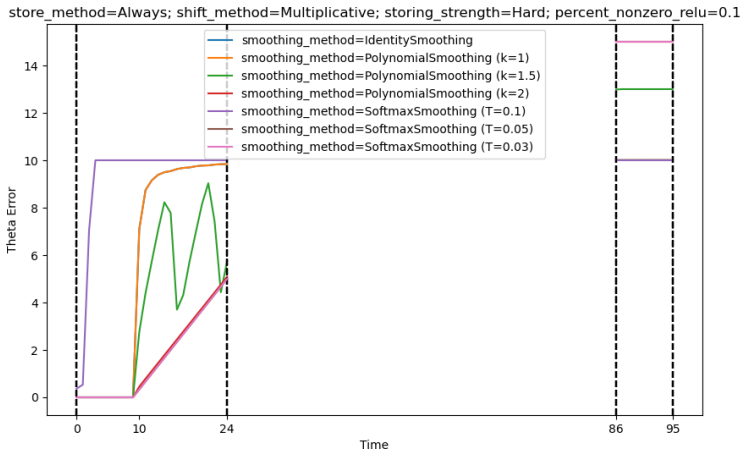

Figure 14: θ dimension error for the AnimalAI test of the RatSLAM algorithm used with various smoothing methods applied after the the Ratshift operation. We notice that softmax smoothing with a low temperature performes best (pink curve). Overall, smoothing methods that caused greater inhibition performed better (higher k values for polynomial smoother and lower T values for softmax smoothing).

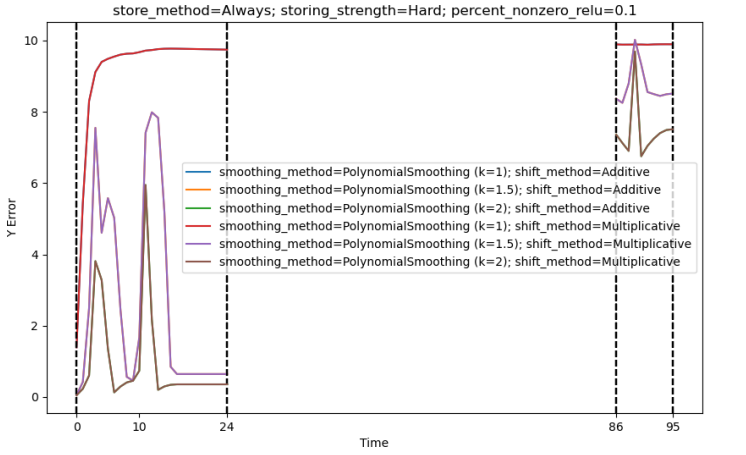

Figure 15: y dimension error for the AnimalAI test of the RatSLAM algorithm used with polynomial smoothing and varying k and shift method. We notice that multiplicative smoothing consistently performed better than additive smoothing.

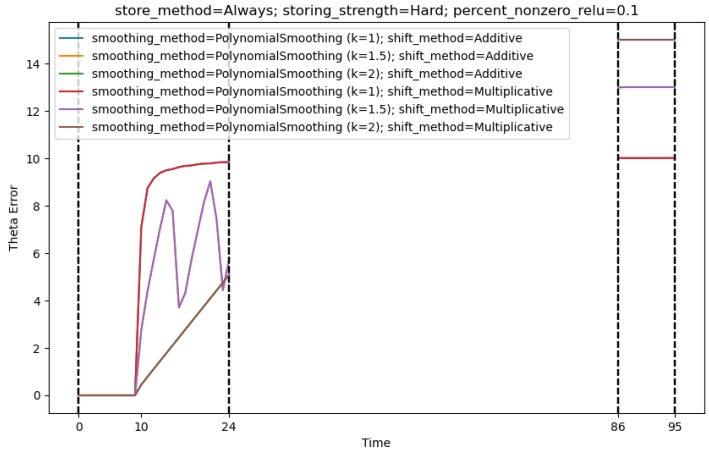

Figure 16: θ dimension error for the AnimalAI test of the RatSLAM algorithm used with polynomial smoothing and varying k and shift method. We notice that multiplicative smoothing consistently performed better than additive smoothing.

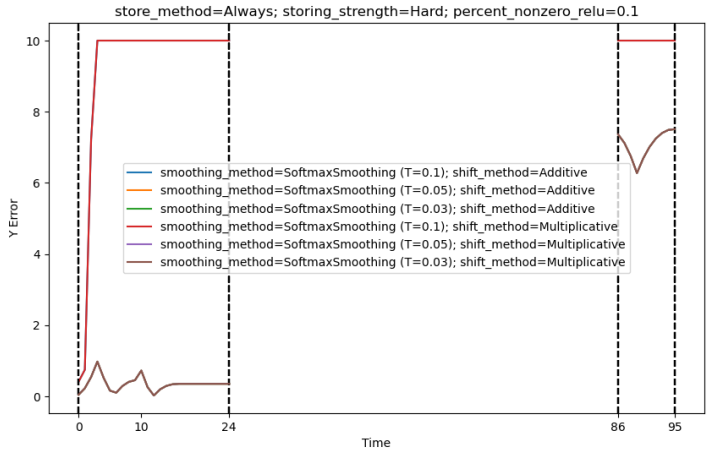

Figure 17: y dimension error for the AnimalAI test of the RatSLAM algorithm used with softmax smoothing and varying k and shift method. We notice no difference between additive and multiplicative smoothing when using softmax smoothing.

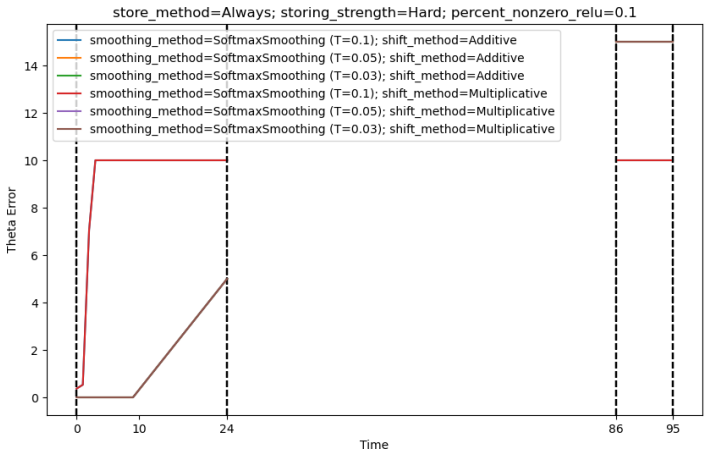

Figure 18: θ dimension error for the AnimalAI test of the RatSLAM algorithm used with softmax smoothing and varying k and shift method. We notice no difference between additive and multiplicative smoothing when using softmax smoothing.

4 Discussion

We successfully generalize the VectorHaSH memory scaffold to three dimensions and non-square modules and experimentally verify that this extended scaffold retains the desirable properties of the original VectorHaSH model, namely that the presence of large basins of attraction and a high number of fixed points when module lengths are pairwise coprime and that the number of hippocampal cells required to achieve full capacity scales linearly with the number of grid modules, not the total capacity or dimensionality of the modules. This is a significant result for scalability, suggesting that the memory capacity can be increased by adding more modules without a proportional increase in the required number of hippocampal neurons.

Furthermore, the choice of initialization method for the grid-to-hippocampal weight matrix was shown to have negligible impact on the average hippocampal neuron activation error, with both standard normal initialization and initialization with a nonzero mean proving suitable for promoting sparsity without sacrificing scaffold properties.

Regarding hippocampal-sensory learning, we find that pseudoinverse learning outperforms Hebbian learning, even with improvements like zero-centering and past input reinforcement. Specifically, Hebbian learning demonstrated insufficient stability to reliably recall previously learned images after a relatively small number of new images were introduced. Moreover, we found that the weights tended to explode without past input reinforcement, and while mean-centering improved stability, it negatively impacted plasticity. In contrast, both analytic and iterative pseudoinverse methods exhibited much greater stability, maintaining good recall performance even after hundreds of images were learned. Therefore, biologically-plausible iterative approaches to pseudoinverse learning are a promising direction for developing stable and high-capacity associative memory in neural models.

When integrating these components into the RatSLAM 2.0 algorithm and testing it in the AnimalAI environment, we observed that the system was performant during periods of continuous odometric input. Different configurations of the algorithm were tested, revealing sensitivity to smoothing methods and shift types on short-term pose estimation accuracy. We find that during the initial phase of movement (before kidnapping), softmax smoothing with a low temperature and a multiplicative shift demonstrated the lowest error in both the y and θ dimensions. Polynomial smoothing also benefited from a multiplicative shift over an additive one, although softmax smoothing's performance was unaffected by the shift type. The choice of hard versus soft storing and the sparsity of the hippocampal layer did not significantly affect the estimated pose error in these tests, likely because their primary impact is on the hippocampal-sensory association learning rather than the immediate grid cell dynamics.

Despite these initial successes in tracking pose during continuous motion, all variations of the RatSLAM 2.0 algorithm tested failed to recover from the simulated kidnapping scenario. This is a significant limitation for a SLAM system. We hypothesize that this failure stems from the agent's strong belief, based on odometric integration, that it is in a location inconsistent with the new sensory input received after teleportation. The current merging and smoothing mechanisms, designed to integrate odometric and sensory information, may effectively suppress the sensory input when it strongly contradicts the odometric prediction, leading to the loss of accurate positional information. The probability mass associated with the true, post-kidnapping location may be zero or near-zero in the odometric belief, causing it to be eliminated by the update rules.

We recognize that our tests had limited scope due to time and computational resource constraints. Longer tests would be valuable to fully assess the long-term accuracy and stability of the system, which might reveal benefits of different smoothing and uncertainty handling mechanisms not apparent in short trials.

Future work should focus on addressing the kidnapping problem. One potential direction is to implement a mechanism for detecting when a kidnapping event might have occurred, perhaps by monitoring the discrepancy between the odometric prediction and the sensory input. Upon detection, the system could prioritize the sensory information, replacing the odometric grid state estimate with the state derived solely from the sensory input to re-localize. Further investigation into alternative merging and smoothing strategies that allow for more robust integration of potentially conflicting information is also warranted. Despite the current limitations in handling kidnapping, the successful generalization of the VectorHaSH scaffold and the promising performance of iterative pseudoinverse

learning represent significant steps towards developing more biologically-plausible and capable SLAM systems.

References

[1] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. Cambridge, Mass.: Mit Press, 2010.

[2] M. Milford, G. Wyeth, and D. Prasser, "Ratslam: a hippocampal model for simultaneous localization and mapping," *International Conference on Robotics and Automation*, Jul 2004.

[3] M. Milford and G. Wyeth, "Mapping a suburb with a single camera using a biologically inspired slam system," *IEEE Transactions on Robotics*, vol. 24, p. 1038–1053, Oct 2008.

[4] E. I. Moser, E. Kropff, and M.-B. Moser, "Place cells, grid cells, and the brain's spatial representation system," *Annual Review of Neuroscience*, vol. 31, p. 69–89, Jul 2008.

[5] S. Chandra, S. Sharma, R. Chaudhuri, and I. Fiete, "Episodic and associative memory from spatial scaffolds in the hippocampus," *Nature*, vol. 638, pp. 739–751, Feb 2025.

[6] T. Mir, P. Yao, K. Duranceau, and I. Prémont-Schwarz, "Are grid cells hexagonal for performance or by convenience?," *Montreal AI and Neuroscience Conference*, Oct 2024.

[7] K. Voudouris, I. Alhas, W. Schellaert, M. Crosby, J. Holmes, J. Burden, N. Chaubey, N. Donnelly, B. Slater, M. Mecattaf G, M. Patel, M. Halina, J. Hernández-Orallo, and L. G. Cheke, "The animal-ai environment: A virtual laboratory for comparative cognition and artificial intelligence research," *arXiv preprint arXiv:2312.11414*, 2024.

[8] J. Tapson and A. van Schaik, "Learning the pseudoinverse solution to network weights," *Neural Networks*, vol. 45, p. 94–100, Sep 2013.

[9] R. C. O'Reilly, "Six principles for biologically based computational models of cortical cognition," *Trends in Cognitive Sciences*, vol. 2, p. 455–462, Nov 1998.

[10] K. V. Mardia and P. E. Jupp, *Directional Statistics*. Hoboken, NJ, USA: John Wiley & Sons, Inc., Jan 1999.

5 Appendix A

5.1 Sparsity calculations

To be an effective scaffold, hippocampal cell activation must be sparse, i.e. the expected percentage of non-zero hippocampal cells active in a fixed point should be low [5]. Let $\psi \in [0, 1]$ denote a target sparsity, with $\psi = 1$ corresponding to the case where no neurons are active.

Sparsity in hippocampal cell activation can be achieved with two methods. The first one, used in VectorHaSH [5], is by sparsely initializing the projection matrix W_{hg} from grid to hippocampal states and using a rectifying function with computing h:

$$(W_{hg})_{ij} \sim \mathcal{N}(0, 1)\text{Bern}(\gamma) \approx \mathcal{N}(0, \gamma) \tag{11}$$

with $\gamma \in [0, 1]$ controlling the sparsity of the matrix. Then h is computed by:

$$h_i = \text{ReLU}\left(\sum_{j=1}^{N_g}(W_{hg})_{ij}g_j - \theta\right) \tag{12}$$

Since there will only be M nonzero entries of g when g is a concatenation of one-hot vectors, we have that

$$h_j \sim \text{ReLU}(\mathcal{N}(0, M\gamma) - \theta) \tag{13}$$

Therefore we wish to find θ such that $P(h_j \leq 0) = \psi$. By using the definition of the normal distribution and standardizing, we have that

$$\psi = \int_{-\infty}^{\frac{\theta}{\sqrt{M\gamma}}} f_Z(z)dz \tag{14}$$

where $Z \sim \mathcal{N}(0,1)$. Hence,

$$\theta = \sqrt{M\gamma}\Phi^{-1}(\psi) \quad \text{or} \quad \theta = \sqrt{M\gamma}\left(\frac{1+\text{Erf}\left(\frac{\psi}{\sqrt{2}}\right)}{2}\right) \tag{15}$$

The second method, that we propose, is to draw entries the projection matrix from a normal distribution with nonzero mean. Given a desired variance σ^2, M modules and a desired ψ sparsity, we have

$$(W_{hg})_{ij} \sim \mathcal{N}(\mu, \sigma^2) \tag{16}$$

Hence,

$$h_j \sim \text{ReLU}\left(\mathcal{N}(M\mu, M\sigma^2)\right) \tag{17}$$

Therefore, to achieve ψ sparsity, we solve for μ:

$$\psi = \int_{-\infty}^{\frac{-M\mu}{\sqrt{M}\sigma}} f_Z(z)dz \tag{18}$$

Hence

$$\mu = -\frac{\sigma}{\sqrt{M}}\Phi^{-1}(\psi) \quad \text{or} \quad \mu = -\frac{\sigma}{\sqrt{M}}\left(\frac{\text{Erf}\left(1+\frac{\psi}{\sqrt{2}}\right)}{2}\right) \tag{19}$$

5.2 Expectation of a rectified normal distribution

Suppose a random variable X is defined $X \sim \text{ReLU}\left(\mathcal{N}(\mu, \sigma^2)\right)$. Note that this is sufficient to handle the case that $X \sim \text{ReLU}\left(\mathcal{N}(\mu, \sigma^2) - \Theta\right)$ since $\text{ReLU}\left(\mathcal{N}(\mu, \sigma^2) - \Theta\right) = \text{ReLU}\left(\mathcal{N}(\mu - \theta, \sigma^2)\right)$.

Then by definition,

$$E[X] = \int_{-\infty}^{\infty} x f_X(x)dx = \int_{-\infty}^{\infty} \text{ReLU}\,(x)\,\frac{1}{\sqrt{2\pi}\sigma}e^{-(\frac{x-\mu}{\sqrt{2}\sigma})^2}dx = \int_{0}^{\infty} x\frac{1}{\sqrt{2\pi}\sigma}e^{-(\frac{x-\mu}{\sqrt{2}\sigma})^2}dx \tag{20}$$

Standardizing, we have that

$$E[X] = \int_{-\frac{\mu}{\sigma}}^{\infty} \frac{1}{\sqrt{2\pi}}(\sigma z + \mu)e^{-\frac{z^2}{2}}dz \tag{21}$$

Then by using standard integration techniques we obtain that

$$E[x] = \mu\Phi^{-1}\left(\frac{\mu}{\sigma}\right) + \frac{\sigma}{\sqrt{2\pi}}e^{-(\frac{\mu}{\sqrt{2}\sigma})^2} = \frac{\mu}{2}\text{Erf}\left(1+\frac{\frac{\mu}{\sigma}}{\sqrt{2}}\right) + \frac{\sigma}{\sqrt{2\pi}}e^{-(\frac{\mu}{\sqrt{2}\sigma})^2} \tag{22}$$

6 Appendix B

6.1 Hyperparameters for experiments

Table 1: Hyperparameters for bidirectional iterative pseudoinverse learning.

ϵ_{sh}	0.1
ϵ_{hs}	0.1
M_{sh}	$1 \cdot$ input neurons
M_{hs}	$1 \cdot$ input neurons

Table 2: Scaffold hyperparameters for hippocampal-sensory layer testing.

$\boldsymbol{\lambda}$	[(3,3,3),(4,4,4)]
N_h	400
Initialization method	Sparsity
ψ	0.9
γ	0.1

Table 3: Hyperparameters for AnimalAI tests.

$\boldsymbol{\lambda}$	[(5,5,5),(8,8,8)]
N_h	600
Initialization method	Sparsity
ψ	0.1
γ	0.1
h-s layer	Iterative pseudoinverse
ϵ_{sh}	0.1
ϵ_{hs}	0.1
M_{sh}	$1 \cdot$ input neurons
M_{hs}	$1 \cdot$ input neurons
σ_{xy}	0.3
σ_{θ}	0.3
Global inhibition constant	0.004
δ_{γ}	1

7 Appendix C

Algorithm 3: Grid-hippocampal scaffold

INITIALIZE MODULES($\boldsymbol{\lambda}$)
  for m *from 1 to* M **do**
    $s_{ijk}^{(m)} \leftarrow \delta_i \delta_j \delta_k \quad \forall (i,j,k) \in \Omega^{(m)}$
  end for
  return $[s^{(1)}, \ldots, s^{(M)}]$

INITIALIZE $W_{hg}(\mu, \sigma, \gamma)$
  $(W_{hg})_{ij} \leftarrow$ Calculated scaling or sparsity **??**
  return W_{hg}

HIPPOCAMPAL FROM GRID(g, θ)
  The suggested value for θ is 0 when W_{hg} is calculated by Equation **??** and the result of Equation **??** when calculated by Equation **??**
  $h \leftarrow \mathrm{ReLU}(W_{hg} g - \theta)$
  return h

GRID FROM HIPPOCAMPAL(p)
  $g \leftarrow \mathrm{ReLU}(W_{gh} h)$
  return g

DENOISE(g)
  $[g_1, \ldots, g_M] \leftarrow g$
  for m *from* 1 *to* M **do**
    $g'_m \leftarrow$ DENOISE(g_m) according to Equation **??**
  end for
  return $[g'_1, \ldots, g'_M]$

ESTIMATE CERTAINTY(s, k)
  `//k = how far from the mean we want to lie between`
  get h from s
  get g from h
  for each distribution $X_{i,d}$, estimate lower bounds $l_{i,d}$ for being within k of the mean using equation 10
  return $\min\{l_{i,d}\}$

Algorithm 4: Iterative Bidirectional Pseudo-Inverse Learning

Input: S, H
Output: Updated weight matrices W_{hs}, W_{sh}

LEARNED_PSEUDO_INVERSE_HS(INPUT, OUTPUT)
  Compute $b_k = (\boldsymbol{\Theta}_{hs} \cdot \text{input}) / (1 + \text{input}^T \boldsymbol{\Theta}_{hs} \text{input})$
  Compute error vector $e_k = \text{output} - W_{hs} \cdot \text{input}$
  Compute normalization factor E
  Compute $\gamma = 1/(1 + (1 - e^{-E})/\epsilon_{hs})$
  Update inhibition matrix:
  $\boldsymbol{\Theta}_{hs} = \gamma(\boldsymbol{\Theta}_{hs} - \boldsymbol{\Theta}_{hs} \cdot \text{input} \cdot b_k^T + \frac{(1-e^{-E})}{\epsilon_{hs}} I)$
  Update weight matrix:
  $W_{hs} = W_{hs} + e_k \cdot b_k^T$
LEARNED_PSEUDO_INVERSE_SH(INPUT, OUTPUT)
  Similar procedure as $learned\_pseudo\_inverse\_hs$ applied to W_{sh}

Algorithm 5: Store memory

STORE MEMORY(g', s)
  if *Hard Storing* **then**
    $g \leftarrow$ DENOISE(g')
  end if
  else
    $g \leftarrow g'$
  end if

Algorithm 6: Inject Velocity Shift into Pose Cells

Input: pose_cells, v, θ, ω, k_x, k_y, k_θ
Output: Updated pose cell activities

new_pose_cells $\leftarrow$ copy of pose_cells
for $g_x \in 0$ *to pose_cells*$_x$ **do**
  for $g_y \in 0$ *to pose_cells*$_y$ **do**
    for $g_\theta \in 0$ *to pose_cells*$_\theta$ **do**
      $\Delta x \leftarrow \lfloor -v\cos(\theta) \cdot k_x \rfloor$
      $\Delta y \leftarrow \lfloor -v\sin(\theta) \cdot k_y \rfloor$
      $\Delta\theta \leftarrow \lfloor -k_\theta \cdot \omega \rfloor$
      $\Delta f_x \leftarrow -k_x v\cos(\theta) - \Delta x$
      $\Delta f_y \leftarrow -k_y v\sin(\theta) - \Delta y$
      $\Delta f_\theta \leftarrow -k_\theta\omega - \Delta\theta$
      Initialize α as a $2 \times 2 \times 2$ zero matrix
      for $i \in \{0, 1\}$ **do**
        for $j \in \{0, 1\}$ **do**
          for $k \in \{0, 1\}$ **do**
            $\alpha[i, j, k] \leftarrow g(\Delta f_x, i) \cdot g(\Delta f_y, j) \cdot g(\Delta f_\theta, k)$
          end for
        end for
      end for
      change $\leftarrow 0$
      for $x \in [\Delta x, \Delta x + 1]$ **do**
        for $y \in [\Delta y, \Delta y + 1]$ **do**
          for $\theta \in [\Delta\theta, \Delta\theta + 1]$ **do**
            change $\leftarrow$ change $+ \alpha[x - \Delta x][y - \Delta y][\theta - \Delta\theta]$
            $\times$pose_cells$[(g_x + x) \mod \text{pose_cells}_x][(g_y + y) \mod \text{pose_cells}_y][(g_\theta + \theta) \mod \text{pose_cells}_\theta]$
          end for
        end for
      end for
      new_pose_cells$[g_x, g_y, g_\theta] \leftarrow$ change
    end for
  end for
end for
return new_pose_cells

Algorithm 7: Interpolation Function g

Input: a, b
Output: Weighting factor

if $b = 0$ **then**
  return $1 - a$
end if
else
  return a
end if

Algorithm 8: Merge methods

ADDITIVE MERGE((g'), $curr$)
  $g \leftarrow g' + curr$ **return** g
MULTIPLICATIVE MERGE((g'), $curr$)
  $g \leftarrow g' \odot curr$ **return** g