

# **A Biologically Realistic Computational Model of the Hippocampus for Reinforcement Learning Applications**

Stephen Polcyn

Advisors: Professor Ken Norman, Dr. Qiong Zhang

## **Abstract**

*This work describes the design and implementation of a biologically realistic neural network simulation of the hippocampus suitable for use with reinforcement learning (RL) agents. The model exposes its key functionality using an HTTP REST API so that any programming language that can make HTTP requests can interact with the model. This new system enables researchers to quickly start using the hippocampus memory model for RL-based, real-time interactive experiments. In the future, we hope to use this model to study the neural underpinnings of the “method of loci,” a highly effective and time-honored memory technique.*

## **1. Introduction**

The “Method of Loci” is an ancient Greek memory technique thought to be first used by poets to memorize long works for performance [25]. The technique’s fundamental idea is to choose a set of “loci” (typically a set of familiar environments, like one’s living room, dining room, and kitchen), define a sequential path through those environments, and then associate items to be recalled with each loci. Items could be things like broccoli or eggs for a grocery list, or a big, block-font number for a list of numbers. Then, to recall the items, one does a sequential, virtual “walk” in their mind through each of these environments, recalling the item bound to each location along the way.

While the days of Greek lyric poets’ memorizing and performing their poetry are long gone, in the modern day, this technique has found a new home with “memory athletes.” These memory athletes are people who train their memory to superhuman levels for competition in annual championships. A famous example is Joshua Foer, who chronicled his training with the method of loci and subsequent

2006 USA Memory Championship win in the New York Times bestseller, “Moonwalking with Einstein” [5].

Despite its storied history and continued modern relevance, the neural underpinnings of the method of loci are still poorly understood. One approach to start understanding the method’s effectiveness is to build computational models of the relevant brain regions. With a biologically realistic computational model, we can study the neural dynamics and performance of the model under varied constraints to gain insight on what could be happening in the brain while the method of loci is in use. Using these models in combination with an RL agent that creates an optimal policy for storing and recalling item-loci pairs further enables us to gain insights about theoretically optimal memory techniques under these biological constraints. In this work, we describe the creation of biologically realistic computational model of the hippocampus that supports real-time interaction with RL agents, enabling RL agents to interact with the hippocampus model as a realistic memory environment in which to create an optimal item-loci encoding policy.

The paper takes the following form: Section 2 describes neuroscience results related to our study, previous work on computational modeling, and the needs of an RL agent. Section 3 describes the design and use of the hippocampus model. Section 4 describes the considerations for creating the RL interface. Section 5 describes the implementation and use of the interface. Section 6 describes the results of a proof-of-concept experiment run using the interface. Section 7 presents limitations and suggestions for future work. Finally, Section 8 presents this work’s conclusions, and Appendix A documents the API in greater technical detail.

## **2. Problem Background and Related Work**

### **2.1. Spatial Memory, Cognitive Maps, and the Hippocampus**

Current research has shown that the method of loci technique is highly reliant on spatial memory, as measured using fMRI data taken from memory champions [14]. Additionally, recent work has demonstrated that location information is particularly discriminable in the brain, suggesting that significant amounts of the brain are dedicated to maintaining and processing spatial context [21].

Since the method of loci is highly dependent on mental navigation of spatial environments, these results together suggest that understanding brain regions involved with navigation of environments will be key to understanding the neural underpinnings of the method of loci.

The hippocampus is exactly one such region. The right posterior hippocampus was one of the regions that Maguire et al. found in their 2003 fMRI study on memory champions and how they use the method of loci, and many other studies have found similar importance for the hippocampus in spatial memory [14, 21, 12, 16]. Taken as a whole, these results show that the hippocampus is essential in the organization and navigation of spatial environments.

The hippocampus has also been shown to be key in the highly related domain of cognitive maps, particularly over shorter timescales [11, 7]. These “cognitive maps” are the brain’s representation of its environment and the relations within that environment, and include not only spatial maps, but also more abstract maps, like social networks [9, 23]. While research has also shown that the hippocampus becomes less involved in the spatial navigation task over longer timescales, experienced memory champions have continued to show activity in their hippocampal regions while using the method of loci, and it is additionally important to understand how the technique builds up representations over time, not just what they look like in their final state [14, 7]. Understanding the building process for the method of loci is particularly important for helping new users of the method to learn quicker and more effectively, as knowledge of what the brain should look like during that time will be critical in diagnosing a user’s progress.

Thus, because the hippocampus is demonstrated to be involved both in both spatial and non-spatial cognitive map navigation and organization, it is logical that any computational method seeking to model a task related to cognitive maps should include the hippocampus. Since the method of loci involves a virtual spatial environment (a clear example of a cognitive map), the hippocampus is a natural choice to model for investigating the method of loci .

## 2.2. Complementary Learning Systems (CLS)

The hippocampus is able to both memorize details of an event and discover patterns across different memories. This dual ability is theorized to be enabled by “complementary learning systems” (CLS) within the hippocampus [15]. In this theory, the hippocampus learns new information rapidly, and then over time it slowly transfers the information to the neocortex, where it is stored for the longer term. A primary difference between the hippocampus and the neocortex is the extent to which the regions allow memories to overlap – in the hippocampus, memories are highly “pattern separated,” whereas in the neocortex, overlapping memory representations are used. Pattern separation is “a process that minimizes overlap between patterns of neuronal activity representing similar experiences,” and it is primarily useful for avoiding confusion between similar memories [13]. Overlapping representations, on the other hand, are useful for finding commonalities across different memories occurring at different times. The CLS theory has been shown over time to provide a fundamentally robust account for many results seen in the literature, indicating that our hippocampus model should reflect this theory [17, 20].

## 2.3. Computational Modeling of CLS

Several recent papers have worked on modeling the hippocampus and its complementary learning systems. The first is a computational model of the hippocampus that incorporates the theta rhythm (a 3-8 Hz signal in the hippocampus) to modulate the strength of different hippocampal pathways, such as those involving the critical CA1 (cornu ammonis 1) and CA3 regions [8, 6]. The main contribution of this model is the addition of error-driven learning, as opposed to exclusively Hebbian learning. The addition of error-driven learning significantly increases the model’s ability to learn and recall abstract patterns, and it is a better overall model of hippocampal dynamics. This model forms the basis for the model used later.

The second work uses a computational model of the hippocampus derived from the first paper’s model to explore more deeply the ways the hippocampus is able to support both rapid statistical learning and rapid episodic memorization [22]. The main contribution of this model is a mechanism

for explaining experimental results that the hippocampus is not only able to identify patterns across memories over long timescales, but also over short timescales. These findings are explained using the different properties of the trisynaptic pathway (TSP) and the monosynaptic pathway (MSP).

The TSP is focused on the “rapid encoding and pattern separation of episodic memories,” i.e., memory storing timestamped details about events in one’s past [8]. The TSP starts with the entorhinal cortex (EC), the part of the neocortex connecting to the hippocampus, then proceeds into to the dentate gyrus (DG), thought to be involved in pattern separation [24], then to the CA3, which also is highly pattern-separated and is thought to store memory traces for episodic memory [8], then finally to the CA1, which is thought to function as an autoencoder for memories and thus is able to reconstruct the original memory from its sparse representation. The TSP is thought to be particularly important in the “pattern completion” task, which is the brain’s ability to reproduce the rest of a memory given only a partial cue. The MSP, on the other hand, composed of just the EC and the CA1, is focused on discovering patterns across memories and learns repeated structures over many experiences. Together, these works contribute a robust hippocampus model and support the Complementary Learning System approach’s validity, providing a solid foundation on which to create a hippocampus model.

## **2.4. The Emergent Neural Network Simulator**

Because this work seeks to build a biologically realistic model suitable for making mechanistic hypotheses about the method of loci, previous work on biologically realistic neural networks is particularly important. The Emergent software is the best example of such work, and it is also the software used for the previously discussed models that studied the CLS theory. The software package has been in active development for over 10 years, and is a “comprehensive neural network simulator that enables the creation and analysis of complex, sophisticated models of the brain in the world” [3]. The software uses the “Leabra” learning algorithm, which stands for “local, error-driven and associative, biologically realistic algorithm” and uses “electrophysiological principles of real neurons” to implement a neural network that behaves like the brain [18]. In addition to the core

code required for creating a model architecture and the Leabra learning algorithm for running it, Emergent also contains several biologically realistic models of different brain processes, including a model of the hippocampus.

## **2.5. RL Needs and Missing Emergent Interface**

A general reinforcement learning setup must have, at a high level, an agent trying to optimize a policy, an environment in which the agent is learning, an action that the agent can take in the environment, and a reward metric to inform it how successful the action was.

Here, the hippocampus takes the role of an RL environment, due to its role as a distinct memory resource providing an intelligent “memory buffer” for the cortex, taking in and processing new memories at a high rate, and then transferring them to the cortex more slowly over time. This difference in function suggests that changing the way memories are presented to the hippocampus versus the rest of the cortex may result in different memory recall accuracy, and thus it should be optimized differently. Accordingly, in this setting, the RL agent takes the role of the brain’s interface to the hippocampus, the entorhinal cortex (EC). While the agent’s precise goal may vary depending on the specific study, in general the RL agent is likely to be seeking a way to present memories to the hippocampus so that later when it (the agent) is queried about a specific memory, it can recall and return the most information from its hippocampus memory storage environment.

In summary, we can consider the reinforcement learning setup to be as follows:

1. **Agent:** The EC connecting to the hippocampus.
2. **Environment:** Computational model of the hippocampus.
3. **Action:** Send partial cue to hippocampus.
4. **Reward:** Metric of recall accuracy.

Thus, a reinforcement learning setup must be able to do two main tasks when interacting with the hippocampus:

1. **Provide a training data set** to the hippocampus model so the model can learn the recall targets.

2. **Query the hippocampus model** with partial cues for the recall targets and obtain a reward metric.

However, while the models included with Emergent’s main code base and other models created with Emergent are usually meant to be designed and then run once to gather statistics, in our research, we are most interested in using the hippocampus model as an interactive environment in which an RL agent can learn an optimal memory policy. The Emergent model does not provide such an interface, so this work addresses that gap by creating a mechanism for real-time interaction between an RL agent and a memory model environment.

## 3. Model Design and Creation

### 3.1. Hippocampus Model

The hippocampus model used is functionally the same as the model available in the `emer/leabra` examples repository [19]. Slight modifications were made to support connecting to the API code, but a primary goal was to make the interface as loosely coupled from the core model code as possible, as this minimizes the work required to update to newer versions of the hippocampus model and ensures new research insights can be quickly integrated into the testing environment. The remainder of this section describes the properties and setup of the model in its current iteration.

### 3.2. Hippocampus Model Setup

**3.2.1. Neural Architecture and Functions** The hippocampus model implements the same Complementary Learning Systems approach as described in Section 2.3. Concretely, this involves an EC layer, a DG layer, a CA3 layer, and a CA1 layer. The EC layer is further divided into the  $EC_{in}$  and  $EC_{out}$ , representing the area of the EC where input is sent to the hippocampal neural network and where the network’s output is received, respectively. The Input layer is used to provide initial activations to the  $EC_{in}$  layer, and thus represents the lower level brain regions (like the visual cortex) that provide inputs to the hippocampus. The connection between the  $EC_{in}$  and  $EC_{out}$  is used to enable “big loop recurrence,” a mechanism which proposes that sending the hippocampal

**Table 1: Connections Between Layers**

Source Layer	Target Layer	Connection Type	Biological Relevance
Input	$EC_{in}$	Feed Forward	Lower Level Brain Regions
$EC_{out}$	$EC_{in}$	Feed Backward	Big Loop Recurrence
$EC_{in}$	CA1	Feed Forward	MSP
CA1	$EC_{out}$	Feed Forward	MSP
$EC_{out}$	CA1	Feed Backward	MSP
$EC_{in}$	DG	Feed Forward	TSP (Perforant Pathway)
$EC_{in}$	CA3	Feed Forward	TSP (Perforant Pathway)
DG	CA3	Feed Forward	TSP (Mossy Fibers)
CA3	CA3	Lateral	TSP (Schaffer Collaterals)
CA3	CA1	Feed Forward	TSP (Schaffer Collaterals)

output stream back into the hippocampus changes its dynamics and enables it to recognize patterns occurring over several memories [10].

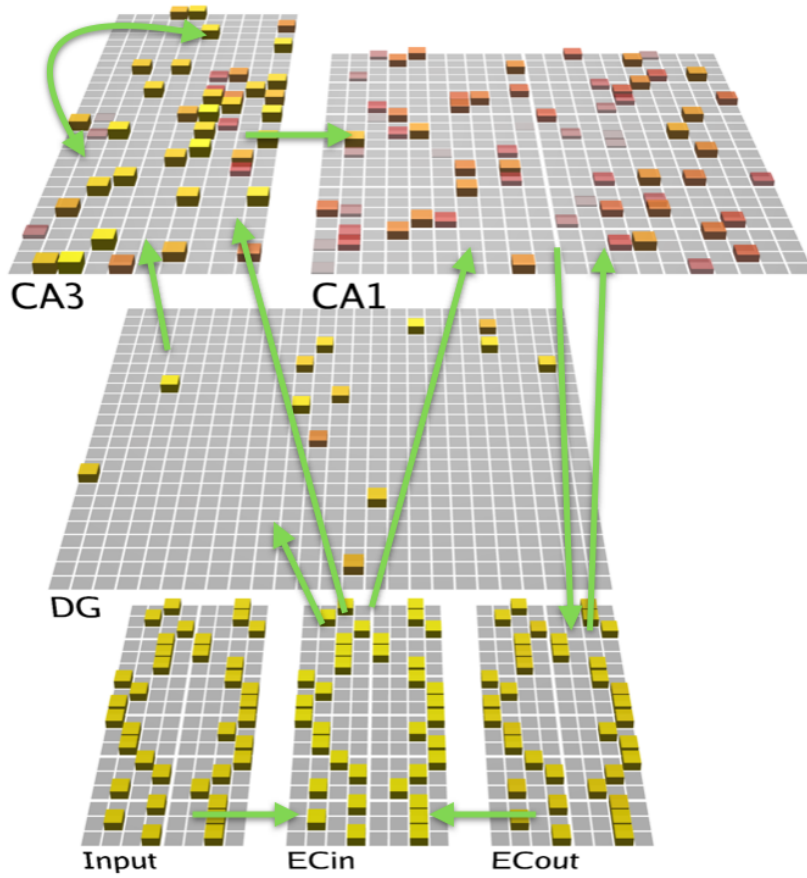
The connectivity between layers is described in Table 1, and a visualization of the full network during training is shown in Figure 1. As described earlier, the layers involved in the TSP are more focused on rapid learning and pattern separation, whereas the layers involved in the MSP are more focused on long-term storage of memory and discovering patterns across them.

**3.2.2. Input/Output Format** The hippocampus model accepts inputs in the form of four dimensional matrices encoding a pattern of activation of neurons. Each four dimensional pattern is composed of a two dimensional matrix of two dimensional “pools,” each of which can be thought of as an individual component of a larger concept represented by the full pattern. An example of such a pattern is shown in Figure 2.

**3.2.3. Model Training** The model uses the input-output pairs to train its weights. The model’s training procedure occurs over several timescales:

1. **Cycle:** One cycle of the Leabra algorithm is executed.
2. **Trial:** In each trial, an individual pattern is presented to the model, and (by default) 100 cycles of the Leabra algorithm are run.
3. **Epoch:** In each epoch, one trial is done for each pattern. Thus, with fifty patterns, one epoch consists of fifty trials.
4. **Run:** In each run, a fixed number of epochs is done. Since the model is re-initialized at the





**Figure 1:** The architecture of the model, showing the Input, EC, DG, CA1, and CA3 layers, as well as the connections between them. A full listing of connections and their types is described in Table 1. The color and height of the squares represents activity, where more height and yellower color means more activity, and the opposite means less activity. Each individual square represents a single neuron, and gray squares are neurons that are inactive during this particular point in time.

beginning of each run, statistics can be averaged across runs to get more reliable performance data.

**3.2.4. Model Testing** After the conclusion of training, the model is ready for testing. Patterns used to test the model are inputs that have been corrupted in some way, usually by removing part of the pattern. The model’s task is then to recall the missing portions. An example of a corrupted pattern is shown in Figure 3. In the method of loci setting, a corrupted pattern might be a pattern that only contains the loci, and the model must then pattern-complete to recall the item portion of the representation.


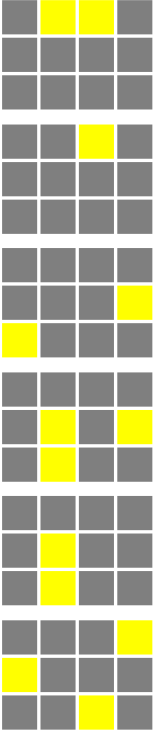
Name	Input	ECout
trn-0 <span>✕</span>		

Figure 2: An example of a pattern used by the model with overall shape (6,2,3,4), translating to “6 rows of 2 columns of 3 row x 4 column pools.” Because the model employs error-driven (i.e., supervised) learning in which input-answer pairs are provided, each training pattern has the input (“Input” column) and the expected output (“ $EC_{out}$ ” column). During training, the model learns to map them to each other. Each square represents an individual neuron, with activated (“on”) neurons shown in yellow, and non-activated (“off”) neurons shown in gray.

#### 4. Interface Requirements and Selection

To expose the hippocampus’ model functionality to an RL agent, there were two primary options:

1. **Tightly integrate an RL agent with the hippocampus model code.** This would require any RL agent to be written in Golang, the implementation language of the model, and require significant time for a new researcher to get up to speed on the model and start programming the required hooks for the RL agent to access model functionality.
2. **Create an Application Programming Interface (API).** The API approach has many advantages:
  - (a) Language-neutral, allowing an RL agent to be written in any language (e.g., Python).

Name	Input	ECout
test-pattern ✕		

**Figure 3: An example of a corrupted pattern. When comparing to Figure 2, one can see the six “slots” in the pattern that are entirely absent of any activity. These are the slots the model must recall (pattern-complete) when presented with this corrupted input pattern.**

- (b) Enables changing model implementation without breaking the interface. This includes expanding the underlying memory system beyond the hippocampus to other brain regions.
- (c) Clearly defines the necessary operations for the model to support, making future development easier in a multi-person research team.

Thus, the API method was chosen as the ideal way to expose the hippocampus’ functionality to an arbitrary RL agent.

## 4.1. Choosing an API Implementation

**4.1.1. GoPy** One API method considered was GoPy [2]. GoPy is software that uses Go’s `cgo` feature to build a C shared library from the original Go code, then generate Python bindings from it. Then, using `CPython`, this shared library can be accessed from within Python. The advantage of this method is that the entire model’s functionality can be accessed in native Python. However, this

method also requires translating the entire hippocampus model from Go into Python, a non-trivial task itself that would additionally require constant revision as updates to the hippocampus model are released.

**4.1.2. Inter-Process Communication (IPC)** Another option was to use some form of IPC, such as shared memory or a message queue, to communicate between an RL agent and the hippocampus model. While this type of implementation is very high performance, which is attractive in a machine learning setting, it also involves considerable complexity and has less platform-independence. Because at this stage of research we didn't see a demonstrated need for the performance of these IPC methods, we chose not to pursue this route and the complexity it entailed.

**4.1.3. Server-Based** The final option considered was an HTTP server-based approach. In this approach, an HTTP server would accept requests, take the appropriate actions on the hippocampus model it was controlling, and then provide the desired response. Because HTTP servers are well-supported within Go and high-quality, well-documented, easy-to-use libraries for interacting with them exist in both Python and Go, this option was highly attractive. Additionally, this makes it possible to interact with the model on a remote server or to easily spin up several models on a local machine to parallelize across models. Thus, due to ease of development and the existence of many well-understood HTTP API architectures, like Representational State Transfer (REST), the HTTP server-based approach using a REST API was selected.

## **5. HTTP REST API Implementation and Execution**

This section describes the technical details of the API's implementations and documents its usage.

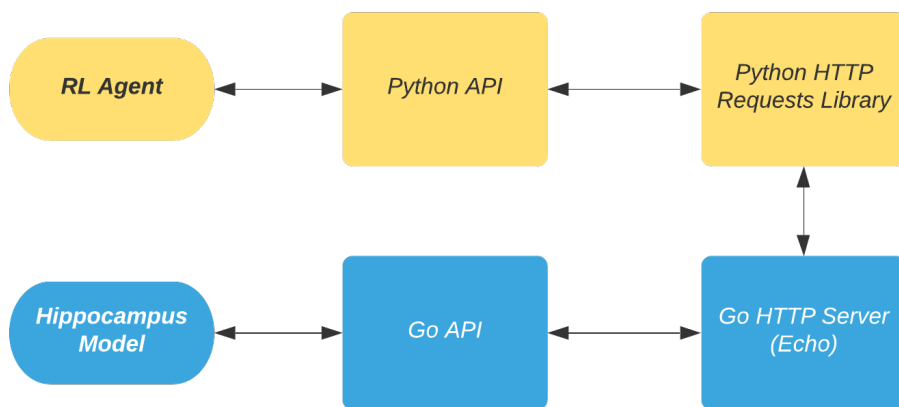
### **5.1. HTTP Server**

The HTTP server was implemented using the Echo framework for Go, a high performance web server that is easy to use, well documented, and feature rich [1]. The HTTP server is started when the hippocampus model is started, and it maintains a reference to the hippocampus model so it can manipulate the model later in response to user requests.

## 5.2. HTTP REST API Documentation

Here, only a high-level overview of the API is presented. For precise documentation of the API, see Appendix A.

The implementation of the REST API handlers is done in Go. Additionally, a simpler Python API that abstracts away the details of formatting inputs and interacting with the HTTP server was written to simplify RL development in Python. A diagram of the path that an API request follows is shown in Figure 4. In total, the API supports three operations which collectively provide the functionality needed for the RL agent to optimize its encoding policy, as well as an additional operation that wraps this functionality into a commonly used function for RL.



**Figure 4: The path a request follows between the RL agent and the hippocampus model. Portions utilizing Python are marked with a yellow background, and portions utilizing Go are marked with a blue background.**

1. **Update Training Patterns** This operation accepts a set of patterns, encoded as a list of  $n$ -dimensional arrays and sets the model's training set to them.
2. **Start Training** This operation starts training the model on its current set of training patterns. The user may specify the number of epochs and runs the model trains for. It waits until the training completes, and returns the total time the training required.
3. **Test Pattern** This operation accepts as input a corrupted pattern and the original/target pattern. It then tests the model on the corrupted pattern, returning the distance between what the model

recalled and the correct pattern, as well as the pattern in the training set closest to what the model recalled.

4. **Step** This operation accepts as input a list of partial cues and a number of times to test each item, and returns the average reward for each partial cue (defined as *total number of neurons in the pattern - neurons recalled incorrectly*). This wraps the **Test Pattern** functionality in a method similar to OpenAI Gym models, and is largely provided as an example for users to write their own `Step` methods.

**5.2.1. Sample API Execution** Here, a typical interaction with an RL agent using the Python API is described, with pseudocode provided.

1. Start model without GUI (to speed up training) from compiled Go binary: `model nogui`
2. Initialize Python API: `api = HipAPI()`
3. Send the training patterns to the model:

- (a) `patterns = randomPatterns()`
- (b) `response, success = api.UpdateTrainingDataPatterns(patterns)`
- (c) <Model parses patterns and updates internal state>
- (d) <Model sends back response>
- (e) `if success: ...`



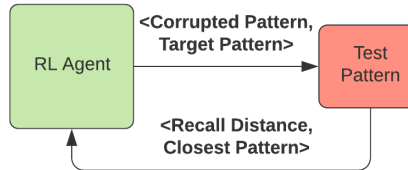
4. With training patterns updated, the model can be trained:

- (a) `response, success = api.StartTraining(maxruns = 1, maxeps = 50)`
- (b) <Model receives request>
- (c) <Model begins training, waiting until completion to return>
- (d) <Model finishes training, sends back total training time>
- (e) `if success: ...`



5. With the model trained, a test can be sent to the model.

- (a) `targetPattern = patterns[0]`
- (b) `partialCue = corrupt(targetpattern)`
- (c) `response, success = api.TestPattern(partialCue, targetPattern)`
- (d) `<Model receives test requests and updates state>`
- (e) `<Model runs partial cue through neural network>`
- (f) `<Model computes distance and closest pattern for recalled pattern>`
- (g) `<Model returns results>`
- (h) `closestPattern = response.json()["ClosestPattern"]["Values"]`
- (i) `if closestPattern == targetPattern: print("Successful Recall")`



## 6. Results

To test the API and model and ensure both remained functional, a demo experiment was run, interacting with the model solely through the API. The experiment tested how the model's recall performance varied as a function of both number of patterns it had to memorize and how corrupted the partial cues were. Additionally, the experiment was run for two different lengths of training epochs, namely `maxepcs = 2, 5`.

Based on the results of Ketz et al., 2013, we expected that recall performance would get worse as more patterns were given to the model [8]. Additionally, we hypothesized that as corruption

increased, model performance should decrease due to less information being present for pattern completion. The results of the experiment, done for corruption ratios between  $[0, 1.0]$  with steps of .1 and total number of patterns between  $[2, 20]$ , with steps of size 2, are shown in Figure 5 and Figure 6. The corruption ratio is defined as the amount of the image that is removed, so a corruption ratio of 0 means nothing is changed, and a corruption ratio of 1 means the entire pattern is changed. The experiment was done with `maxruns = 1`, `maxepcs = 2, 5`. Further increasing the training epochs could increase training accuracy, especially for larger lists, at the cost of training time. Modulating training time is one way to model how well a method of loci user has learned their memory palace and the items within it.

Indeed, exactly as expected from the 2013 paper and our hypothesis, memory recall does get worse as the number of patterns increases, and it also gets worse as the corruption ratio increases. The best results are obtained for very few patterns and very low corruption, and the worse results are obtained for the highest list sizes and corruption ratios. Additionally, increased training length resulted in increased accuracy, with the model trained for five epochs exhibiting lower performance drops as corruption ratio increased and as the number of patterns increased.

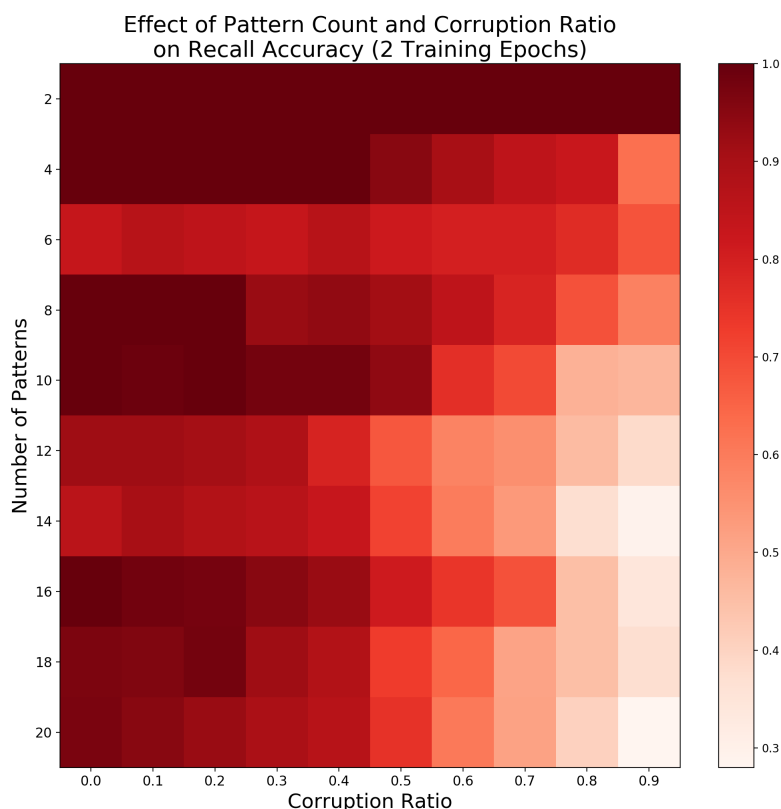
In the method of loci context, we would expect the results in the corruption ratio of .5 column to be the most representative of how the hippocampal model will perform, as the model will be given half of the pattern to work with (the loci part). However, it is also possible the RL agent could decide to allocate more of the space in the pattern to the loci (especially when the list of items to remember is shorter), in which case a lower corruption ratio may more accurately characterize the method of loci performance.

## 7. Limitations and Future Work

### 7.1. Limitations

Despite the API's suitability for running experiments, it does still have a few limitations. One limitation is that the API does not give full access to every method of the hippocampus model. This is by design, as it reduces the amount of code necessary to maintain and it lowers the amount of

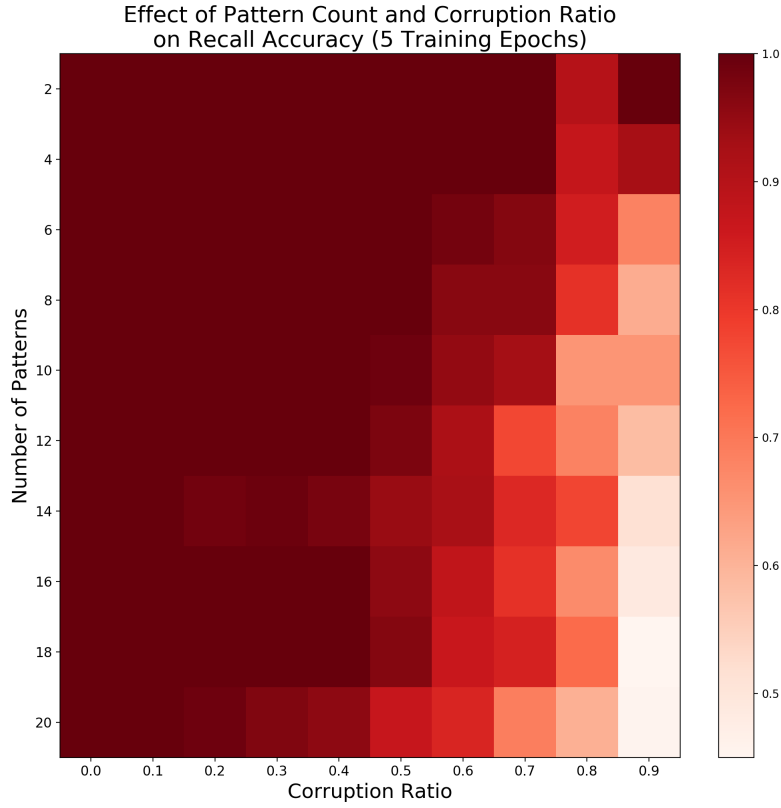




**Figure 5: The model’s recall performance as a function of both number of patterns and corruption ratio after being trained for two epochs.**

information that new users of the API need to learn before they can start using it. However, there may exist use cases down the road where researchers want finer-grained access to the model. In this case, it is relatively easy to add new API methods and endpoints for specific methods and functionality. Additionally, the model can be opened using its GUI to configure specific parameters, and the parameters can be saved to a file and loaded at runtime, reducing the necessity of an ability to access many parameters through the API.

Another limitation is that the model currently uses a fixed pattern size of 6x2x3x4, and thus only accepts patterns of exactly that size. This is the default pattern size for the Emergent hippocampus model and is determined by the size of its input layer. Setting the network size and parameters at runtime is a possibility and is intended to be supported by the API in the future. However, changing the network size may also have unexpected implications on the performance of the model, and



**Figure 6: The model’s recall performance as a function of both number of patterns and corruption ratio after being trained for five epochs.**

assessing this is beyond the scope of this paper.

## 7.2. Future Work

In addition to resolving some limitations of the model, there are several interesting areas of potential future work. One is to improve the communication speed and efficiency between the models. While profiling did not reveal significant time being spent on sending the patterns over HTTP or parsing them once they arrived at the model, it is possible that for much larger pattern sizes or numbers of patterns the cost could become non-negligible.

Another performance-related area of future work is increasing hardware utilization while training the model, either through increased parallelization with CPU or by developing a way to train the model on a GPU. One potential way to increase parallelism would be to run each pattern for an

epoch in a distributed fashion, aggregating the weight updates to the neural network at the end of the epoch. The implications for the model’s learning rate are, however, unclear, and experiments on the best aggregation method would be important to assess the impact of such distributed training.

A more practical and easy way to increase performance and experimental capabilities is to design a mechanism to launch several versions of the model at once, thereby taking advantage of model parallelism. This would enable much quicker experiments if running multiple datasets. For example, one could spin up ten instances of the model, then run experiments involving 10-100 patterns (step size of 10) all at the same time, one on each model.

Finally, extending the current hippocampus model to become a more generalized memory model (e.g., by including models of other brain regions involved in memory), would offer a more robust overall account of how method of loci technique dynamics evolve over time and in different brain regions. This could also provide insight on the findings that long-term cognitive maps rely less on the hippocampus [7].

## **8. Conclusions**

Taken as a whole, this work demonstrates a novel interface for exposing a biologically realistic hippocampus model’s functionality to a general reinforcement learning agent. It uses an HTTP REST API that can be interacted with from virtually any language, due to the wide proliferation of libraries to interact with web servers. Additionally, we present a Python API that further simplifies the API by wrapping the necessary logic for formatting the data and making the HTTP requests, enabling RL agents written in Python (a very common language for research and ML) to even more quickly get working with the hippocampus model. Experiments run using the API demonstrate it is fully capable of running experiments and additionally show that the model’s results concord with existing literature on computational hippocampus models.

### **8.1. Acknowledgments**

This work was not done alone. I would like to thank my advisor Professor Ken Norman for identifying this project as highly aligned with my interests, connecting me to it, and then discussing

productive ways in which I could contribute. I’d also like to thank him for his consistent help in connecting me with key resources and determining high-level directions and goals of the project.

I would also like to thank my postdoctoral advisor, Dr. Qiong Zhang, who has been with me every step of the way, discussing new and sometimes confusing results, contextualizing my work within the larger study, providing key feedback and suggestions to ensure my project was as successful and productive as possible, and helping me think about how this work and research might fit into future plans.

Finally, I would like to thank Randy O’Reilly, author of the Emergent software, for his assistance in understanding the model, and Lucy Wang, for statistical assistance that provided a key insight on some long-unexplained results.

## References

- [1] Echo - High performance, minimalist Go web framework. <https://echo.labstack.com/>.
- [2] Go-python/gopy. go-python, May 2020.
- [3] B. Aisa, B. Mingus, and R. O’Reilly. The emergent neural modeling system. *Neural Networks: The Official Journal of the International Neural Network Society*, 21(8):1146–1152, Oct. 2008.
- [4] H. Eichenbaum. The role of the hippocampus in navigation is memory. *Journal of Neurophysiology*, 117(4):1785–1796, Apr. 2017.
- [5] J. Foer. *Moonwalking with Einstein: The Art and Science of Remembering Everything*. 2012. OCLC: 965138568.
- [6] M. E. Hasselmo, C. Bodelón, and B. P. Wyble. A proposed function for hippocampal theta rhythm: Separate phases of encoding and retrieval enhance reversal of prior learning. *Neural Computation*, 14(4):793–817, Apr. 2002.
- [7] M. Hirshhorn, C. Grady, R. Rosenbaum, G. Winocur, and M. Moscovitch. The hippocampus is involved in mental navigation for a recently learned, but not a highly familiar environment: A longitudinal fMRI study. *Hippocampus*, 22(4):842–852, Apr. 2012.
- [8] N. Ketz, S. G. Morkonda, and R. C. O’Reilly. Theta coordinated error-driven learning in the hippocampus. *PLoS computational biology*, 9(6):e1003067, 2013.
- [9] R. Kitchin. Cognitive Maps. In N. J. Smelser and P. B. Baltes, editors, *International Encyclopedia of the Social & Behavioral Sciences*, pages 2120–2124. Pergamon, Oxford, Jan. 2001.
- [10] R. Koster, M. J. Chadwick, Y. Chen, D. Berron, A. Banino, E. Düzel, D. Hassabis, and D. Kumaran. Big-Loop Recurrence within the Hippocampal System Supports Integration of Information across Episodes. *Neuron*, 99(6):1342–1354.e6, Sept. 2018.
- [11] D. Kumaran and E. A. Maguire. The Human Hippocampus: Cognitive Maps or Relational Memory? *Journal of Neuroscience*, 25(31):7254–7259, Aug. 2005.
- [12] C. T. Kyle, J. D. Stokes, J. S. Lieberman, A. S. Hassan, and A. D. Ekstrom. Successful retrieval of competing spatial environments in humans involves hippocampal pattern separation mechanisms. *eLife*, 4:e10499, Nov. 2015.
- [13] A. D. Madar, L. A. Ewell, and M. V. Jones. Pattern separation of spiketrains in hippocampal neurons. *Scientific Reports*, 9(1):1–20, Mar. 2019.
- [14] E. A. Maguire, E. R. Valentine, J. M. Wilding, and N. Kapur. Routes to remembering: The brains behind superior memory. *Nature Neuroscience*, 6(1):90–95, Jan. 2003.
- [15] J. L. McClelland, B. L. McNaughton, and R. C. O’Reilly. Why there are complementary learning systems in the hippocampus and neocortex: Insights from the successes and failures of connectionist models of learning and memory. *Psychological Review*, 102(3):419–457, July 1995.
- [16] D. Nielson, T. Smith, V. Sreekumar, S. Dennis, and P. Sederberg. Human hippocampus represents space and time during retrieval of real-world memories. *Proceedings of the National Academy of Sciences*, <http://www.pnas.org/content/early/2015/08/12/1507104112.abstract>, Aug. 2015.

- [17] K. A. Norman and R. C. O'Reilly. Modeling hippocampal and neocortical contributions to recognition memory: A complementary-learning-systems approach. *Psychological Review*, 110(4):611–646, 2003.
- [18] R. C. O'Reilly. Leabra. <https://grey.colorado.edu/emergent/index.php/Leabra>.
- [19] R. C. O'Reilly. Emer/leabra. emergent, May 2020.
- [20] R. C. O'Reilly, R. Bhattacharyya, M. D. Howard, and N. Ketz. Complementary Learning Systems. *Cognitive Science*, 38(6):1229–1248, 2014. \_eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1551-6709.2011.01214.x>.
- [21] J. Robin, B. R. Buchsbaum, and M. Moscovitch. The Primacy of Spatial Context in the Neural Representation of Events. *Journal of Neuroscience*, 38(11):2755–2765, Mar. 2018.
- [22] A. C. Schapiro, N. B. Turk-Browne, M. M. Botvinick, and K. A. Norman. Complementary learning systems within the hippocampus: A neural network modelling approach to reconciling episodic memory with statistical learning. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 372(1711):20160049, Jan. 2017.
- [23] R. M. Tavares, A. Mendelsohn, Y. Grossman, C. H. Williams, M. Shapiro, Y. Trope, and D. Schiller. A Map for Social Navigation in the Human Brain. *Neuron*, 87(1):231–243, July 2015.
- [24] A. Treves, A. Tashiro, M. P. Witter, and E. I. Moser. What is the mammalian dentate gyrus good for? *Neuroscience*, 154(4):1155–1172, July 2008.
- [25] F. A. Yates and F. A. Yates. *The Art of Memory*. Number v. 3 in Selected Works / Frances Yates. Routledge, London ; New York, 1999.

## 9. Appendix A: Technical Documentation

This appendix provides finer-grained documentation of the technical details of the HTTP REST and Python APIs.

### 9.1. HTTP REST API Documentation

#### 9.1.1. Update Training Patterns

1. URL: `/dataset/train/update`

2. JSON Body Parameters:

- (a) **Source:** Either “file,” if reading the dataset from an Emergent formatted CSV file, or “body” if the new patterns are transmitted in the HTTP request body.
- (b) **Filename:** If using the “file” method, then the path to the CSV file containing the new dataset, in Emergent format.
- (c) **Patterns:** If using the “body” method, then the array of patterns that should be uploaded into the model. These patterns must be the same shape as the input layer of the model, hold only `int` values, and values may be only 0 or 1. The patterns additionally must be uploaded in JSON array format, and the string representation of the 4-D array should use parentheses to represent subarrays. For example, a 2x2 array would be as follows: “((1,2), (3,4))”, and

two 2x2 patterns would be encoded in JSON as “[‘((1,2), (3,4))’, ‘((1,2), (3,4))’]”. The use of parentheses is to avoid confusion with JSON array syntax.

- (d) **Shape:** The shape of the pattern, e.g., “(2,2)”. All patterns must be the same shape as the network’s input layer.

3. Returns: An HTTP status code signaling the success/failure of the request.

## 9.2. Start Training

1. URL: `/model/train`

2. JSON Body Parameters:

- (a) **MaxRuns:** The number of runs to run the model for. Generally, this should be 1, as the model’s training state is reset in between runs.

- (b) **MaxEpcs:** The number of epochs to run during each run. This translates to how many times each pattern is presented to the model during training (each epoch presents every pattern once).

3. Returns: An HTTP status code, and a string with information about how long training took and the run/epoch parameters used.

## 9.3. Test Pattern

1. URL: `/model/testpattern`

2. JSON Body Parameters:

- (a) **Shape:** The shape of the pattern, e.g., “(2,2)”. The pattern must be the same shape as the network’s input layer.

- (b) **CorruptedPattern:** The corrupted pattern’s representation. All requirements for patterns presented in Section 9.1.1 also apply here.

- (c) **TargetPattern:** The target pattern that should be recalled by the model when given the corrupted pattern. All requirements for patterns presented in Section 9.1.1 also apply here.

3. Returns: An HTTP status code, the distance between the recalled pattern and the target pattern

(Hamming distance), and the closest pattern in the training data to the recalled pattern (as measured by Hamming distance).

## 9.4. Python API Documentation

To simplify development for the end user, an API that wraps interacting with an HTTP API in simple Python methods was written. That Python API follows the same structure as the HTTP API, but has simplified arguments and return values.

### 9.4.1. Update Training Patterns

There are two ways to update patterns:

To update patterns from a file:

1. Python Method: `UpdateTrainingDataFile(self, filename)`
2. Parameters:
  - (a) **filename:** The path to the CSV file containing the new dataset in Emergent format, as a string.
3. Returns: A tuple with the HTTP response and a boolean indicating whether the request executed correctly.

To update patterns from Numpy arrays:

1. Python Method: `UpdateTrainingDataPatterns(self, patterns)`
2. Parameters:
  - (a) **patterns:** The patterns, as a Python list of 4-D Numpy arrays containing only `int` values (i.e., `dtype=int`) that are either 0 or 1.
3. Returns: A tuple with the HTTP response and a boolean indicating whether the request executed successfully.

## 9.5. Start Training

**Note:** This function waits until training is complete to return, so it is expected if it takes a while to return.

1. Python Method: `StartTraining(self, maxruns, maxepcs)`
2. Parameters:

- (a) **MaxRuns:** The number of runs to run the model for. Generally, this should be 1, as the model's training state is reset in between runs. Default value: 1.
- (b) **MaxEpcs:** The number of epochs to run during each run. This translates to how many times each pattern is presented to the model during training. Default value: 50.
- 3. Returns: A tuple with the HTTP response (which contains information about training time within the body) and a Boolean indicating whether the request executed successfully.

## 9.6. Test Pattern

- 1. Python Method: `TestPattern(self, corruptedPattern, targetPattern)`
- 2. Parameters:
  - (a) **corruptedPattern:** A 4-D Numpy array representing the corrupted pattern (partial cue), with entries of type `int` that are either 0 or 1.
  - (b) **targetPattern:** A 4-D Numpy array representing the target pattern (full memory), with entries of type `int` that are either 0 or 1.
- 3. Returns: A tuple with the HTTP response (which contains both the Hamming distance between the recalled pattern and the target pattern and the closest pattern in the training data to the recalled pattern) and a Boolean indicating whether the request executed successfully.