A Macrocolumn Architecture Implemented with Temporal (Spiking) Neurons

Version 0.0

J. E. Smith

06/25/2022

Abstract

With the long-term goal of reverse-architecting the computational brain from the bottom up, the focus of this document is the macrocolumn abstraction layer. A basic macrocolumn architecture is developed by first describing its operation with a state machine model. Then state machine functions are implemented with spiking neurons that support temporal computation. The neuron model is based on active spiking dendrites and mirrors the Hawkins/Numenta neuron model. The architecture is demonstrated with a research benchmark in which an agent uses a macrocolumn to first learn and then navigate 2-d environments containing randomly placed features. Environments are represented in the macrocolumn as labeled directed graphs where edges connect features and labels indicate the relative displacements between them.

1. Introduction

Viewed from a computer architect’s perspective, the brain’s neocortex can be modeled as a hierarchy of abstraction layers (Figure 1). The overarching objective of the research considered in this paper is to reverse-architect the computational brain from the bottom up, beginning with biologically plausible neuron models, in order to discover the way it performs cognitive computation. As part of this bottom-up approach, this paper is focused on the green-shaded layers at the lower end of the architecture stack. The objective is to define and connect these three architecture layers in a biologically plausible manner and to demonstrate the way they perform basic cognitive functions.

Figure 1. An architecture for the computational brain consists of abstraction layers. This paper is focused on the bottom three layers with the model neuron layer being further subdivided into three sublayers.

To date, the line of research conducted by the author has been primarily targeted at the neuron and minicolumn layers. Prior work most relevant to the discussion here can be found in [35][36][37]. This document is focused on the next layer in the hierarchy: the macrocolumn.
A promising approach to macrocolumn architecture is advocated by Jeff Hawkins and his colleagues at Numenta [14]-[18],[22], and this paper marks a transition in the author’s research toward the Hawkins/Numenta approach to neuron operation and macrocolumn design. This is reflected in Figure 1 where the neuron layer is subdivided into three sublayers. By incorporating the concept of “active dendrites” [15], the author’s “spiking neuron” in prior work is relegated to a spiking dendritic segment in this paper. A dendrite is then composed of multiple segments, and a neuron is composed of multiple dendrites. As part of the transition to active dendrites, neuron inputs are categorized as being “distal” and “proximal”, each playing a role in neuron operation.

As put forward by Mountcastle [29], the basic macrocolumn structure is uniform across the neocortex. Although specific connectivities and neuron characteristics may differ across regions, the operating principles remain the same. Hawkins [18] accepts the Mountcastle uniformity hypothesis and focuses on principles of functional uniformity:

1) Reference frames are present everywhere in the neocortex.
2) Reference frames are used to model everything we know, not just physical objects.
3) All knowledge is stored at locations relative to reference frames.
4) Thinking is a form of movement.

The Mountcastle uniformity hypothesis and Hawkins’s four principles underpin the modeling approach in this document.

Hodgkin and Huxley [19] developed their famous neuron model by studying squid giant axons because their size makes them good candidates for physical experimentation. Although their research focused on a particular type of easy-to-study neuron, the fundamental principles apply to all neurons. Similarly, in this document, a basic cognitive architecture is constructed and applied to an easy-to-understand illustrative benchmark that involves movement and feature learning in a physical two dimensional space. For this application, the reference frame is a two dimensional grid, and knowledge is the presence of features at grid locations. From a researcher’s perspective this 2-d grid application is easy to understand, explain, and simulate.

However, “thinking is a form of movement” applies to all forms of thinking including higher level thinking, not just navigation of physical environments (see [17], p. 11). As two examples, “thinking” includes mathematical reasoning and language processing. Taken together, the uniformity hypothesis and “thinking is a form of movement” strongly suggest that the same structure and principles displayed in the 2-d grid-based application developed in this paper may apply, at a basic level, to all forms of thinking.

1.1 System Architecture

Macrocolumns are developed and studied as part of an overall system organization depicted in Figure 2. An agent generates actions that cause movement in an environment. As movement occurs, a macrocolumn consisting of grid and place cells associates relative grid locations with environmental features, signaled via sensations.

The model agent invokes both internal and external movements. If the agent is directing movement through a physical environment, then an external movement will be paired with an identical internal movement, thereby allowing the macrocolumn to track the movements internally. However, in some instances the agent may invoke an internal movement only. It might do this when performing “mind travel” [33], for example, when the agent wants to be aware of features within its current environment without actually performing the physical movements. Based on the outcome of such mind travel the agent may or may not proceed with the actual external movement. Finally, as noted above, the same basic system architecture can be applied to non-physical environments. In that case, the only movements may be internal movements.
In the system architecture proposed here, the macrocolumn does not make movement decisions, rather the agent does, so the macrocolumn plays a role loosely analogous to a Q-Table in reinforcement learning. As the agent directs movement through an environment, the macrocolumn dynamically updates synaptic weights that collectively characterize environmental features and their relative locations as they are observed. After adequate learning, the synaptic weights, through an inference process, provide information to the agent as it tries to accomplish a specific objective, e.g., inferred displacements and environment identifiers (inferred envIds) in the figure.

Figure 2. The basic system architecture contains macrocolumn(s) that support an agent which issues movement commands as it navigates a grid-based physical environment. For a given environment (envId in), the agent learns an association between an externally sensed feature, the previous feature, and the relative displacement (Δx and Δy) between the two. The previous feature and displacement are maintained by a grid subsystem within the macrocolumn. Then, when queried by the agent (by providing a target feature) the macrocolumn returns the inferred displacement between the current feature and the target feature. This displacement allows the agent to move directly from its current location to a target feature’s location via both internal and external movement signals. Internal movement affects the internal grid subsystem, and external movement affects the location in the physical environment. Internal movements track external movements by design.

1.2 Research Benchmark

The objective here is to demonstrate macrocolumn architecture and function by constructing a system as shown in Figure 2 containing a single macrocolumn. A demonstration system capable of executing a benchmark suggested by Lewis et al. [22] is designed and verified via simulation.

The research benchmark involves a mouse navigating dark 2-dimensional environments (in the dark only features local to a given location can be sensed). Because a single macrocolumn can hold multiple environments at the same time, the mouse is introduced to a sequence of environments, say a few tens, and has the opportunity to explore and learn each of them. As initial explorations take place, the macrocolumn learns spatial relationships among features within each of the environments.

Then, the mouse is dropped into a random environment at a random location. It first orients itself by moving about in the environment, associating features it senses with what it previously learned. Assuming no inherent ambiguity in the environments, this process eventually converges to a unique feature within a known environment: the mouse is oriented.
One of the features may be “cheese”, so if the mouse is placed in a learned environment in a “hungry” state, after it has oriented itself, the mouse’s internal agent uses the macrocolumn to navigate to the cheese. The agent implements the navigational strategy. In this paper, a naive agent is used to illustrate the basic ability to navigate (no cheese is involved). A smarter agent that can learn to find cheese via reinforcement learning is not within the scope of this paper.

Figure 3. "Mouse in the Dark" A mouse first learns multiple environments, each sprinkled with randomly placed features. After learning, when placed at a random place in a random environment, the mouse first orients itself by identifying its environment and then uses learned features to navigate the environment, for example to find a specific feature such as a piece of cheese.

To summarize, the macrocolumn supports three basic tasks under the direction of an agent:

1) Exploration: It can learn environments by encountering features while exploring.
2) Orientation: When placed in an arbitrary learned environment, it can orient itself.
3) Navigation: After orientation has taken place, it can navigate the environment to achieve goals.

This paper constructs a macrocolumn architecture composed of biologically plausible spiking neurons that provides these three functions in a straightforward manner.

1.3 Contributions

The main contributions are the following.

1) A state machine description of macrocolumn operation. The state machine constructs a directed graph representation of environments that is stored in an associative memory.

2) The implementation of spiking neurons based on active dendrites. These neurons provide the same computational capabilities as Hawkins/Numenta neurons.

3) Construction and demonstration of a macrocolumn that implements the state machine description with spiking neurons. This is done by developing models for dendritic segments, dendrites, neurons, mini-columns, and macrocolumns. It is proposed that the macrocolumn uses loops formed by autaptic connections as a non-synaptic way of maintaining state for extended periods of time.
2. State Machine Description

In this section, a state machine description of macrocolumn operation is developed. This description is easy to understand and is a good starting point for detailed macrocolumn implementation. First the research benchmark problem is described. Then, the state machine description is given in some detail, using the research benchmark as an extended illustrative example.

2.1 Research Benchmark

An environment, labeled “α” is shown in Figure 4a. A mouse (controlled by the agent) is dropped into the environment at an arbitrary location (“start” in Figure 4b) and begins exploring the environment at the agent’s direction. The agent explores randomly, encountering features along the way. As it does so, it keeps track of its movements by maintaining the net displacements (∆x and ∆y) from the most recent feature it has encountered. When it reaches the next feature, it uses this information to construct and learn an edge in a directed graph representation of the environment (Figure 4c). During exploration it may encounter the same feature more than once, so a given feature in the graph may have indegree > 1. After an adequate amount of learning, the mouse is introduced to a second environment β which it also learns. Here, “adequate amount of learning” is loosely defined. At a minimum it should learn a connected graph, so at least every feature is reachable from every other feature. On the other hand, a complete graph is likely to be too expensive except for the simplest environments.

This process is repeated for a second environment β shown in Figure 5. In Section 0, a total of 40 environments, each 30×30, are simulated.

![Figure 4. a) The environment α encompasses a 15x15 grid containing randomly placed features A through E. The coordinates of each feature are shown. b) Beginning at an arbitrary starting point, the mouse explores the environment. c) As it explores, it learns the features and their relative locations by constructing a directed graph labeled with displacements from one feature to another. In general, the same feature can be visited more than once.](image-url)
Figure 5. A second environment, \( \beta \).

After the initial exploration/learning process is complete, the mouse is dropped into a random environment at a random location. It orients itself by moving about the environment, associating movements and features it senses with the previously learned graphs. As long as the environments are theoretically resolvable using the learned information, this process eventually converges to a unique feature within a known environment: the mouse is oriented.

After orientation, the agent can use the learned information to navigate the environment. If the agent is located at a given feature and applies a desired destination (target) feature to the macrocolumn, the macrocolumn will yield the displacement to the destination feature as an output – provided the macrocolumn previously learned the labeled edge between the two features. This displacement information allows the agent to move directly to the desired destination feature.

2.2 State Machine Implementation

The state machine implementation maintains a state vector that is constantly updated as the agent directs movement through an environment. The state machine consists of 1) a state vector, 2) state vector update functions, and 3) an associative memory that stores learned state vectors. The implementation has three operating modes:

- **Exploration**: the agent moves the state machine around the environment, and the state machine constructs (learns) the directed graph environment model as it goes. The directed graph is stored as a set of state vectors held in the associative memory.

- **Orientation**: beginning at an unknown environment and location, the agent moves over the grid and queries the associative memory as part of a process that determines its environment and location.

- **Navigation**: the agent queries the associative memory for displacement information that allows it to move directly and efficiently within the environment.

In discussions to follow, the modes are described and illustrated separately. However, in practice, they are not mutually exclusive. For example, exploration can occur concurrently with navigation.
**State Vectors**

As the agent moves through an environment, a [state vector] consisting of five components is maintained within a macrocolumn. The state vector contains all information necessary for constructing labeled directed edges in an environment graph. The components are:

- **envId**: the environment identifier
- **tail**: the feature at the tail of the directed edge.
- **Δx**: the x dimension displacement between **tail** and **head**
- **Δy**: the y dimension displacement between **tail** and **head**
- **head**: the feature at the head of the directed edge.

A state vector is written as: [envId tail Δx Δy head]. An example state vector taken from environment α in Figure 4 is: [α C 4 4 B]. At any given time some, or all, of the vector components may be null, for example the two displacements are null in state vector [α C − − B]. Furthermore, some components may be ambiguous; for example envId is ambiguous in [α,β C 4 4 B].

**State Machine Inputs**

- **envId_in** – from the agent when in learning mode; otherwise null
- **target** – from the agent when in navigation mode
- **xmove** – x dimension movement, from the agent – internal movement in Figure 2
- **ymove** – y dimension movement, from the agent – internal movement in Figure 2
- **feature** – feature at the current grid location (if there is one) from the environment

**State Machine Outputs**

- **i_envId** – inferred environment, to the agent – inferred envId in Figure 2
- **i_Δx** – inferred Δx, to the agent – inferred displacement in Figure 2
- **i_Δy** – inferred Δy, to the agent – inferred displacement in Figure 2

**Associative Memory**

Because the associative memory is implemented as place cells in the macrocolumn, it is denoted here as PL. As the agent explores the environment to learn the graph representation, the associative memory PL stores edges in the form of state vectors:

\[
PL ← [\text{state vector}].
\]

This is essentially a form of one-shot learning – it is a simple write to the associative memory. However, it is important to note that in the macrocolumn implementation in Section 6, PL learning is a biologically plausible, incremental update process that accounts for both the current state and what has been learned from previous states. This is the only significant difference between the state machine description and the spiking neuron macrocolumn implementation.

Based on what it has learned, PL can be queried to yield three inferred outputs: i_envId, i_Δx, and i_Δy. To query the PL, an input key is applied. The key is essentially a state vector, where the head component of the key must be non-null, and the other components may or may not be null. To form a query key, the head component is provided by the agent via signal target. Other components of the key come from the current state vector.

\[
[key] = [envId \ tail \ Δx \ Δy \ target]
\]

When the key is applied to PL, it produces outputs from the PL entry whose components best match the key’s components.

\[
i_{envId}, i_{Δx}, i_{Δy} ← PL (key)
\]

Where i_envId is the envId component of the stored state vector, i_Δx is the Δx component, and i_Δy is the Δy component. The i_envId output is used during the orientation process, and the inferred displacements are used during the navigation process.
Update Functions
State vector update functions for each of the components of the state vector follow:

envId: if exploration == 1 envId ← envId_in
if orientation == 1 envId ← i_envId ← PL(key)
if navigation == 1 envId ← envId

tail: if head ~ null tail ← head
else tail ← tail

Δx, Δy: if feature == null, Δx ← Δx + xmove; Δy ← Δy + ymove
else Δx ← null, Δy ← null

head: head ← feature

Output Functions
The output envId comes directly from the state vector. Outputs i_Δx and i_Δy are produced by querying the associative memory when in navigation mode:

if navigation == 1 i_Δx, i_Δy ← PL(key)

2.3 Detailed Example

Exploration
Figure 6 illustrates a sequence of movements during the exploration of environment α. Initially, all state vector components except envId are null. The agent provides envId_in, the identifier for the environment about to be learned. As the agent moves through the grid, no state vector updates occur until the first feature, C, is reached. Whenever the current grid location contains a feature, the displacements are initialized to Δx, Δy = [-,-], so the state vector becomes [α - - - C]. Next, there is a single time step pause (zero movement). At the beginning of the pause step, head ~ null, so tail ← head, and at the end of the step, C is at both the head and the tail, and the relative displacements are null.

Then, the agent moves away from the grid location containing C onto a location with no features, so head ← null and tail ← tail. The movement [1,1] yields state vector [α C 1 1 -]. After some additional moves, feature B is reached. In the example sequence, this is shown as a single move [3,3]. At that point the state vector components are all non-null: [α C 4 4 B], and the state vector is a complete description of the edge connecting C and B in environment α. The presence of a complete state vector triggers its storage in the associative memory PL.

Exploration continues in the same manner, and PL stores each new edge as it is discovered. Similarly, the agent explores environment β. After exploration, the contents of PL are shown in Figure 7.
State Vector Sequence: Exploration

| movement | state vector | comments |
|----------|--------------|----------|
|          | α - - - - -  | initial state |
| -1 1     | α - - - C   | first feature reached |
| 0 0      | α C - - C   | pause; grid cells auto-reset |
| 1 1      | α C 1 1 -  | move & track displacement |

3 3 α C 4 4 B feature B reached – store vector
0 0 α B - - B pause; grid cells auto-reset
-2 -3 α B -2 -3 - mid-way to next feature
-3 8 α B -5 5 A feature A reached – store vector
13 -6 α A 8 -1 D feature D reached – store vector
-8 8 α D 0 7 E feature E reached – store vector
-7 -8 α E -7 -1 C feature C reached – store vector

Figure 6. Exploration of environment α. The blue arrows indicate the state update sequence: given a state vector, a movement is applied, and this leads to the next state. PL contents are in no particular order to emphasize associative addressing.

| PL Contents |
|-------------|
| α B -5 5 A |
| β A -9 -4 C |
| α E -7 -1 C |
| β B -6 1 D |
| α D 0 7 E |
| β E 3 4 A |
| β C 4 4 A |
| α A 8 -1 D |
| β D 8 -5 E |
| α C 4 4 B |
| β B 5 0 A |

Figure 7. Contents of PL after both environments α and β have been explored.
Orientation and Navigation

Assume orientation begins with PL contents as in Figure 7. The orientation sequence is in Figure 8. Say the mouse is placed in the $\beta$ environment at location [5, 5]. At that point, it knows nothing regarding its whereabouts, so the state vector is completely null. It then begins moving about the environment like it did during exploration. To shorten the example, say that it happens to make the move [-1,1] and encounters feature C. There is a single step pause, and PL is queried with key = [null null C]. Because the proximal input is non-null, this key is sufficient trigger the query $i_{envId} \leftarrow$ PL(state vector). All PL entries with C as the head match the key equally well. Consequently, in this two environment example both envIds become part of an ambiguous state vector $[\alpha, \beta \ C - \ C]$.

The orientation process continues through a series of movements summarized by the movements [-2,-2] and [6,6] so the net movement is [4,4], and feature B is encountered. When the PL is queried with $[\alpha, \beta \ C 4 4 \ B]$, it happens that both environments have exactly the same edge, so PL again returns an ambiguous envId. The orientation process continues, and after a net movement of [-6,1] a third feature D is reached, and the state vector becomes $[\beta \ B - 6 \ 1 \ D]$. This time when PL is accessed, the envId is unambiguous because only environment $\beta$ has a completely matching edge. At this point orientation is achieved. The agent knows it is in environment $\beta$ at feature D.

After a one step pause, navigation begins. The state vector is $[\beta \ D - \ -]$, and the agent performs a PL query to infer displacements that will take it to feature E. The agent sets target $\leftarrow$ E, and $i_{\Delta x}, i_{\Delta y} \leftarrow$ PL([$\beta \ D - - E$]). The best match yields $i_{\Delta x} = 8$, and $i_{\Delta y} = -5$. As can be seen in Figure 5, these displacements will, in fact, take the agent to feature E.

| movement | state vector | comments |
|----------|--------------|----------|
| - -      | - - - - -    | initial state – state vector null |
| -1 1     | - - - - C    | first feature reached; environment is ambiguous |
| 0 0      | $\alpha, \beta \ C - - C$ | pause, access PL |
| -2 -2    | $\alpha, \beta \ C -2 -2$ | perform random movement to a featureless grid cell |
| 6 6      | $\alpha, \beta \ C 4 4 \ B$ | continue until second feature reached; environment is ambiguous |
| 0 0      | $\alpha, \beta \ B - - B$ | pause, access PL |
| -6 1     | $\beta \ B -6 1 \ D$ | third feature reached; environment is $\beta$; orientation complete |
| 0 0      | $\beta \ D - - D$ | pause – then navigate to feature E |
| 0 0      | $\beta \ D - - E$ | Apply feature E to PL to get inferred $\Delta x, \Delta y$ |
| 8 -5     | $\beta \ D 8 -5 \ E$ | inferred displacement becomes next movement |

Figure 8. Example of orientation and navigation. Bold underlined values are inferred.
3. Implementation: Coding and Synchronization

Given the state machine description, the next few sections lay out a biologically plausible spiking neuron-based architecture that implements a version of the state machine and associative memory.

3.1 Temporal Coding and Communication

Values are encoded as spike volleys. A volley $a$ consisting of $p$ spikes is denoted as: $a = [a_1, ..., a_p], a_i \in \{0, 1, ..., t_{\text{max}}, \text{-} \}$, where the null symbol “-” indicates the absence of a spike. In Figure 9, $a = [0, 3, -1]$. These values indicate spike times with respect to the beginning of a $\gamma$ cycle, so the length of a $\gamma$ cycle is $t_{\text{max}} + 1$ time units. When performing temporal functions, the null value has properties one would normally ascribe to infinity, i.e., it is greater than all the other values [34].

A useful interpretation of spike time encoding is that the spike time indicates the relative strength of the signal, i.e., how strongly the neuron producing the spike was stimulated. As in the case of the biological neocortex, spike time resolution is quite low, so the small integer values in Figure 9 are representative of an actual implementation.

A state vector is represented as a spike volley containing 5 bundles.

In this document, volleys on bundles are often binarized, i.e., all spikes occur at the same time. However, as part of temporal neuron processing, output volleys are temporal and indicate relative strengths as noted in the above paragraph. For example, $[0-0-0-0]$ and $[-2-2-2-]$ are binarized volleys.

A state vector is represented as a spike volley containing 5 bundles. As an example, say there are 4 environments, 4 features, and an 8×8 grid. If the state vector components are $\text{envId} = 3; \text{tail} = 2; \Delta x = 4; \Delta y = 3; \text{head} = 4$, an encoding is:

$[---0-0--0-2-3---4-0-0]$.  

Observe that the specific state vector components are encoded only by the presence or absence of a non-null value at a given position in a bundle. The numerical value in the encoding is the component’s relative strength expressed as a spike time. For example, the 2 in the 4th position in bundle 3 indicates $\Delta x = 4$, and the relative strength is 2. An encoding of the same state vector with different strengths is:

$[---1-1--3-3-3-4-0-0]$.  

3.2 Synchronization

High level neural operation is consistent with Abeles’s synchronous synfire chains [1]. Neurons are organized in layers (“pools” according to Abeles) and processing proceeds as waves of spikes (volleys) pass from one layer to the next in a synchronized fashion. In a given volley, there is at most one spike per line.
There are two levels of synchronization in the model architecture. The model is synchronous with discrete steps, and within a step, computation is further divided into time units.

At the neuron level, activation potentials are modeled as pulses synchronized with a discrete unit time clock. The intent is that the resolution of the discrete unit time clock matches the (relatively low) precision of biologically-based signals. The unit time clock is used for simulation and is projected to be used in a direct CMOS implementation. In an implementation, whether simulation or hardware, a “spike” is a single time unit pulse that occurs within a volley. The unit time at which the spike occurs encodes its relative value with respect to other spikes that may occur within the same volley.

A gamma clock marks off cycles that correspond to steps. A gamma cycle consists of multiple time units and is the total time it takes to transmit a spike volley to an excitatory neuron plus the time it takes for the synapses and neuron body to generate an output spike (if any) followed by a pass through winner-take-all (WTA) inhibition (not shown explicitly in the figure). In contrast to the unit time clock which is only present in the model and not in the biology, a gamma cycle is present in the biological neocortex. WTA inhibition coincides with the inhibitory portion of the gamma cycle [5]. In the remainder of the paper “time” refers to unit time; “step” refers to a gamma cycle.

**Figure 10.** An Abeles synfire chain organizes neurons (circles) into layers, separated by synaptic crossbars (small triangles). Spike volleys pass from one layer to the next, synchronized via gamma cycle inhibition. Each volley contains at most one spike per line. As pointed out by Abeles, the layout in the figure does not represent the anatomy, rather it represents the processing time sequence. Hence, the figure is drawn in an unrolled form for illustrative purposes; in reality, there can be any amount of feedback among layers. For example, the red neuron shown at two different points in the chain is the same physical neuron. (Figure taken from Scholarpedia [1])
3.3 Macrocolumn Overview

Figure 11 is a system schematic that consists of a macrocolumn, an agent, and a model environment. The macrocolumn’s function is divided into the following sub-functions.

**tail loop** – implements the *tail* update function.

**Grid cells (GD)** – implements the 2-d grid and path integration. Also included is the *displacement loop*.

**Place cells (PL)** – implements the associative memory.

**envId loop** – implements the *envId* update function.

The gamma cycle boundary is indicated with a heavy dashed line. The small S blocks select from one of two inputs, controlled by the agent. For the operations that depend on the mode, these select the mode-dependent inputs. Depending on whether the mode is exploration or navigation, one of them selects the PL proximal input and the other selects envId. The *min*/WTA blocks are described in the following section.

Figure 11. System schematic. Agent and Environment blocks are part of the simulation framework.
4. Tail Loop

The tail loop (Figure 12) maintains the tail state as the agent navigates the environment. It implements the function:

\[
\text{if } \text{head} \neq \text{null} \quad \text{tail} \leftarrow \text{head} \\
\text{else} \quad \text{tail} \leftarrow \text{tail}
\]

This function also serves as a simple example of temporal processing. It includes a non-synaptic way of capturing and maintaining state, and it employs WTA inhibition.

![Diagram of autaptic tail loop](image)

Figure 12. Tail loop implemented with autaptic neurons. In this network, the head and tail bundles are size \( n \).

4.1 State: Autaptic Loops

The tail value must be maintained as state for a variable number of gamma cycles as the agent travels from one feature to the next. A guiding principle of biologically plausible spiking neural computation is that synapses hold static state, i.e., state that persists for relatively long periods of time (many gamma cycles). Using synaptic state to track the tail state variable (as well as some other state variables) does not appear to be a good fit, however.

There is another way to implement long term state, however. It is well documented that a significant number of neuron connections are autaptic [32] – that is, the axon of a neuron feeds back to one of its own synapses. This forms an autaptic loop that provides a mechanism for maintaining state that appears to be suitable for holding state variables. The autaptic tail loop is shown in Figure 12.

4.2 Inhibition

In the model described in this paper, inhibitory neurons are not modeled individually. Rather they are modeled as a bulk WTA process. WTA methods go far back in the annals of machine learning. Thorpe [38] first proposed that WTA inhibition should be performed on a bundle of lines carrying temporally encoded spikes.

A WTA inhibition block’s input and output bundles have the same number of lines. An input volley on the input lines is translated to an output volley containing only the first spike(s) from the input lines. That is, the input spike with the earliest spike time is passed through uninhibited as an output spike; all others yield no output spike.

Tie cases occur relatively frequently because the model operates on very low precision integers. In this paper, two methods of handling ties are employed, although an eventual research objective is to reduce this to one.

With \( t\text{-WTA} \) all the tying first spikes are passed through to the WTA outputs. The \( t\text{-WTA} \) method is both simple to implement and biologically plausible. This method is used in the autaptic tail loop.
A second WTA method is used in later sections; for completeness it is defined here. This method, 1-WTA breaks ties and selects a single “winning” spike. The tie breaker first selects spikes coming from the excitatory neuron with the highest body potential at the time the spike is generated, and any remaining ties are broken systematically by selecting the neuron with the lowest index (in the simulation model). The objective is to give the same result whenever the same set of input spikes result in a tie; using the lowest index is a simple way of achieving this objective.

### 4.3 Tail Loop Operation

The operation of the tail loop is described beginning with the output of the normalize block. The normalize block is present in simulation to model the effect of the \( \gamma \) cycle, and it outputs a binarized tail with temporal value 0. For example, if the current tail is the fourth of five features (D), then it encoded as the volley: \([- - - 0 -]\). Proceeding around the tail loop, the +2 temporal delay element produces the binarized volley \([- - - 2 -]\). A set of \( n \) parallel excitatory neurons merely relay the input volley. These are the neurons with autaptic connections.

The head input comes directly from the current feature and is encoded with a 1-hot binarized volley with the temporal value 1. For example, the second feature (B) is represented as \([-1- ---]\).

The temporal \( \min \) performs a spike-wise function on its inputs: \( \min ([-1---], [-2--]) = [-1-2-] \). Finally, the tWTA block selects the earliest spike(s), keeping all the ties. In this example, the winner is unique: \([-1- ---]\).

Continuing on, this volley passes through normalization, yielding the volley \([-0---]\), and after the fixed delay of 2 it is: \([-2---]\). If there is no input feature during the next step, i.e. the head input is \([------]\), the tail volley of \([-2---]\) will continue going around the loop. As long as head remains null the same tail value will circulate around the loop indefinitely. Eventually, if a non-null head value comes along with a binarized volley of time 1, then the combination of the \( \min \) function and tWTA will replace the current tail, and the new tail will be captured in the loop.

### 5. Grid Cells and Path Integration

In the application being considered here, the agent invokes actions that cause physical movement through a 2-d environment. Simultaneously, it invokes analogous internal movement, and through the mechanism of path integration, the GD cells use these internal actions to track movement within an internal grid representation.

Because “all thinking is movement”, when higher level thinking is taking place, the only movement may be internal. Furthermore, even when navigating a physical environment, internal movement can be invoked without the corresponding external actions. For example, virtual movement or “mind travel” [33] can be used by an agent to plan future actions to achieve some objective.

The application of grid cells and path integration to high level thinking is clearly a very challenging problem. By choosing the problem of physical navigation through 2-d space, the big challenges are admittedly being sidestepped in favor of demonstrating the basic elements of a complete working system.

As illustrated in Figure 11, grid cells and path integration are implemented with their own nearly autonomous state machine. Conceptually, this mechanism acts as a sort of trip odometer that is reset every time a feature is currently being sensed and, as movement occurs, it tracks net relative displacements until the next feature is reached, at which time it resets until the next feature is reached, etc. The displacement state is maintained in an autaptic loop.
5.1 Grid Cells

For the research application being studied here, there are two environments: an external physical one and an internal one in the macrocolumn grid used for tracking movement. The fact that both can be described as 2-d spaces greatly simplifies the implementation.

Note that the biological grid system appears to be triangular [28], possibly for physical reasons: macrocolumns can be packed more tightly if a triangular grid is used. However, for the problem at hand there does not appear to be any computational advantage in using triangular grids, so a Cartesian system is chosen here to simplify the implementation.

In the external space considered in the research benchmark, locations and displacements are specified in a two-dimensional coordinate system (although the concepts extend to a space of any number of dimensions and/or coordinate system.) Agent-invoked movements through the environment occur in the external space. Specifically, external movements are specified as changes in external space coordinates.

Displacements in the internal space maintained in GD are encoded as two spike volleys, each corresponding to one of the dimensions. Agent actions (the 2-d movements) are also specified with two volleys. Both displacements and movements are specified with one-hot encodings. For example, in an 8×8 system, the displacement [3,5] is represented as a binarized [-0-----|-----0-----].

5.2 Path Integration

An important observation [18] is that all biological macrocolumns implement an internal grid-based navigation system that appears to be effectively hard-wired. That is, it appears that an innate grid organization squeezes through the “genomic bottleneck” [42] and is a built-in mechanism throughout the neocortex. Some characteristics may differ from one region of the neocortex to the next, but the working hypothesis is that in every region there is an innate grid capable of holding frames in one way or another.

With one-hot encoded volleys, movements appear as shifts (left for a negative movement, right for a positive movement), so in the model the GD cells operate functionally as shifters. Although the GD cells can be implemented via neurons (see Appendix), in the model studied here the shift function is implemented directly. In the 2-d application, there are two shifters, one for each dimension. Shifts wrap-around so the grid becomes a biologically plausible 2-d torus [33].

The capabilities built into the specific shifter(s) depend on the modeled movements. For example, if movements are restricted to at most one grid cell per step, then only shifts of -1, 0, +1 are implemented. For the 2-d system considered here, shifts of any number of grid cells are allowed. This makes motion (and simulations) faster.

5.3 Model Implementation

Recall that in the state vector implementation, Δx and Δy update functions are:

\[
\begin{align*}
\text{if } \text{feature } & \text{ is null } \quad \Delta x & \leftarrow \text{null} \quad \Delta y & \leftarrow \text{null} \\
\text{else} \quad \Delta x & \leftarrow \Delta x + x\text{move} \quad \Delta y & \leftarrow \Delta y + y\text{move}
\end{align*}
\]

Referring to Figure 11, the GD block consists of two wrap-around shifters where the shift counts are applied to the P input and the displacements to be shifted are applied to D inputs\(^1\). The shift output is held in an autaptic loop which feeds back to the inputs.

When there is a feature at the current grid location, the displacements are initialized to null. As movement occurs from a grid cell containing a feature to featureless grid cells, shifts of Δx and Δy track the agent-directed movements (xmove and ymove) by performing what are essentially unary additions.

\(^1\) “P” and “D” stand for proximal and distal inputs. They are defined below for excitatory neurons, and their application to GD is in the Appendix.
6. Place Cells

Place cells implement the associative memory contained in a macrocolumn.

6.1 Excitatory Neuron Model

The spiking excitatory neuron model is patterned after the active dendrite HTM neuron described in Hawkins and Subutai [15]. The implementation defined here is a biologically plausible spiking (temporal) version that, although not exactly equivalent to the HTM implementation, provides the same features and capabilities.

Briefly, the neuron body is fed by a number of dendrites, and each dendrite is composed of a number of segments, each of which receives a number of distal inputs (see Figure 13). Also associated with each of the dendrites is a proximal input. Although one could use more than one proximal input per dendrite, in the model here there is only one. Synapses join the inputs to the dendrite, and each synapse has an associated weight. After learning, synaptic weights tend to be bimodal: weights are either 0 or a maximum weight, $w_{\text{max}}$.

![Figure 13. Spiking excitatory neuron with active dendrites. The dark synapses have maximum weight, and the lightly shaded synapses have zero weight. Input spikes on $a$, $c$, and $A$ will yield an early spike on output $y$ indicating a good pattern match. An input spike on $A$ only (or input spikes on $b$, $d$, and $A$) will yield a later output spike indicating a weaker pattern match; Input spikes on $a$ and $c$ without a spike on proximal input $A$ yield no output spike.](image)

The biological rationale articulated in [15] is that a single spike arriving at a distal synapse is unable to propagate along the dendrite and increase a neuron’s body potential by any significant amount. Rather, it takes multiple spikes on distal synapses arriving at about the same time in close spatial proximity (within the same dendritic segment) to bring the neuron’s body potential to a level where the neuron is primed to spike. Even then a spike on a proximal synapse is required to trigger an output (axon) spike from the neuron body.

In general, a spike on a proximal input is a necessary and sufficient condition for generating a neuron output spike. Given a proximal input spike, the output spike occurs earlier or later, depending on how many distal spikes arriving at a segment match maximum weight synapses. The more matches having maximum weight, the earlier the output spike.

As a general interpretation, the pattern of distal input spikes defines a specific context, and a proximal spike indicates a specific feature that is present at the specified context. In the model defined here, distal inputs are divided into bundles with each bundle being associated with some part of the context.
For the mouse-in-the-dark benchmark, there are four distal input bundles: \textit{envId}, \textit{tail}, \Delta x, and \Delta y. The proximal input is \textit{head}, the current feature. Following subsections describe excitatory neuron operation in more detail.

**Dendritic Segments**

If PL is viewed as an associative memory that stores state vectors, a dendritic segment holds a single state vector. A dendritic segment generates spikes according to a spike response model (SRM0) initially proposed for complete neurons [8]. Figure 14 illustrates SRM0 operation as applied to a single active dendritic segment. Synapses map input spikes to response functions, depending on the weight.

![Figure 14. Example of dendritic integration for a single segment. Distal inputs $a_1$ - $a_4$ and proximal input $b_1$ are communicated over a bundle of four lines and a single line respectively. Spike times, i.e., the values, are relative to the beginning of a gamma cycle. As spikes arrive at the synapses, weights determine the amplitudes of response functions, and the response functions are shifted according to the relative spike times. At the dendrite’s integrator, the time-shifted response functions are summed to yield a potential for the given segment. If there is a proximal spike and the potential reaches the threshold value $\theta$ an output spike is produced on output $z_1$.](image)

The response function for a proximal synapse is generally greater than response functions for the distal synapses. In this paper, the response function for proximal synapses is given explicitly, and the response functions for the distal synapses are the proximal functions reduced by a factor of $D$, where there are $D$ distal bundles. Typically, the number of distal bundles is small. In the research benchmark there are four.

By choosing the response functions, a designer can influence the dendrite’s functional characteristics. Specifically, the \textit{ramp no leak} response function $\rho(w,t)$ maps an integer weight $0 \leq w \leq w_{\text{max}}$ and integer time $t$ onto the non-negative integers:

$$
\rho(w,t) =
\begin{cases} 
0 & \text{if } t < 0 \\
= t + \lfloor w/w_{\text{max}} \rfloor & \text{if } 0 \leq t < w \\
= w & \text{if } w \leq t
\end{cases}
$$

For the illustrative benchmark used in this paper, the slope of the response functions is increased to two to speed up simulations:

$$
\rho(w,t) =
\begin{cases} 
0 & \text{if } t < 0 \\
= 2 \times (t + \lfloor w/w_{\text{max}} \rfloor) & \text{if } 0 \leq t < w \\
= w & \text{if } w \leq t
\end{cases}
$$

For example, if $w_{\text{max}} = 8$, response functions are given in Figure 15.
Figure 15. Ramp no leak response functions used in illustrative benchmark.

A dendritic segment’s contribution to its neuron’s body potential is the sum of spike-time-shifted response functions. For segment $j$ having $p$ distal inputs $a_{1-p}$, and a single proximal input $b_j$, the potential $v_j$ for a segment with $D$ distal bundles is computed as:

$$v_j(t) = \rho(w_j, t - b_j) + \sum_i \rho(w_{ij}, t - a_i)/D$$

Where $w_j$ is the weight of the proximal synapse and $w_{ij}$ is the weight of distal synapse $i$ connected to segment $j$.

A segment forms spike-time-shifted response functions and sums them to form $v_j(t)$ as described above. Depending on the segment’s potential (and assuming there is a proximal spike), the output is produced via the spiking function $\sigma$:

$$z_j = \sigma(v_j(t), \theta) = \text{the smallest } t \text{ for which } v_j(t) \geq \theta;$$
$$z_j = \text{null if there is no such } t.$$  

Observe that unlike the step response functions commonly used in integrate-and-fire (IF) and leaky integrate-and-fire (LIF) neuron models, a ramp-integrate-and-fire (RIF) neuron has response function with a sloping leading edge that provides greater temporal processing capabilities (Figure 16). This is not a new observation – in a 1997 paper by Maass [24] a response function with a sloping leading edge is shown to be computationally more powerful than a response function with a step leading edge. Finally, leak is not part of the response function because it is asserted that the primary function of the leak is to perform a reset; in a silicon version (and in simulation), a global reset at the end of every gamma cycle serves this function.
Inference

Recall that as an associative memory, PL is accessed with a state vector that acts as a key. Some of the key components may be null, but the head must be non-null. Given a key, the output consists of certain components of the stored state vector that best matches the key.

\[ i \Delta x, i \Delta y \leftarrow PL(key) \]
\[ i_{envId} \leftarrow PL(key) \]

In the first case, used for making navigational enquires, the envId, tail, and head components of the key are non-null. The displacements of the best matching state vector are returned as outputs. In the second, used during orientation, the envId of the best matching state vector is returned.

During inference, determining and outputting the strength of the best key match is implemented via the response functions. This process is explained for the research benchmark.

Example

In the research benchmark, state vectors applied to excitatory neurons are \[ [envId \ tail \ \Delta x \ \Delta y \ head] \], where head is a proximal input, and the others are distal inputs. Consider inference occurring during navigation (see the last entry in Figure 8). The agent is in a known envId \( \beta \) and a known tail \( D \); the displacements are null. The agent would like to find the displacement to feature E. Consequently, it sets target to E and applies the key: \( [\beta \ D \ - \ - \ E] \) to the neuron inputs.

Figure 16. Comparison of neuron models. Values are highlighted to show functional differences. a) The classic McCulloch-Pitts neuron [26] computes the dot product of a binary input and a weight. If the sum exceeds the threshold, a binary 1 is output. In the example, the input value on \( a_1 \) differs between the top and bottom inputs. However, the threshold is crossed regardless, so the output is the same. The two input patterns are not distinguished. b) An IF neuron (an LIF neuron has the same leading edge behavior) with binarized spiking inputs. The function is the same as the McCulloch-Pitts neuron. c) An RIF with binarized inputs. With a ramp leading edge (slope == 1), the neuron is able to distinguish the difference in inputs. A spike on input \( a_4 \) yields an output spike one time unit earlier than the case where there is no spike on \( a_4 \).
Assume all input spikes are binarized and occur at $t = 0$. Learning has been completed and all weights are at either 0 or $w_{\text{max}}$. The proximal response function is shown in Figure 15, although only the functions for 0 and $w_{\text{max}}$ are actually used in this example. The spiking threshold $\theta = 8$. Table 1 shows the body potential as a function of time for different numbers of distal input matches, i.e., where there is a spike input to a synapse with maximum weight. Green shaded table entries indicate the body potential at the first time the threshold is crossed. For a proximal spike plus 4 distal matches (row P+4D), the body potential reaches the threshold at time $t = 1$. For P+3D and P+2D, the threshold is reached at $t = 2$, for P+1D and P+0D, $t = 3$. The more distal matches there are, the earlier the output spike time.

Table 1. Response Functions and Body Potentials for Combinations of Synapse Matches

|            | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
|------------|----|----|----|----|----|----|----|----|----|
| P+ 4D      | 4  | 8  | 12 | 16 | 16 | 16 | 16 | 16 | 16 |
| P + 3D     | 3.5| 7  | 10.5| 14 | 14 | 14 | 14 | 14 |
| P + 2D     | 3  | 6  | 9  | 12 | 12 | 12 | 12 | 12 |
| P+ 1D      | 2.5| 5  | 7.5| 10 | 10 | 10 | 10 |
| P+ 0D      | 2  | 4  | 6  | 8  | 8  | 8  | 8  |

In this example, the best matching state vector in the associative memory (Figure 7) is $[\beta, D, 8, -5, E]$, which has two distal matches and will yield a spike at time 2. All the other state vectors have at most one distal match and will yield spikes at time 3.

**Dendrites**

A dendrite is composed of multiple dendritic segments, all enabled by the same proximal input. An example dendrite composed of three segments is shown in Figure 17. Consequently, these correspond to state vectors with the same head; in terms of the environment graph, these are all edges coming into the same head node. Because of the $\text{min}$ function, the quality of the best match among the segments is passed through to the output $z$.

![Figure 17. a) A dendrite composed of three segments. For modeling purposes, the dendrite’s proximal synapse is replicated, so summations of distal and proximal response functions occur concurrently rather than as two separate steps. The function, however, is the same. b) Symbol for dendrite functional block.](image-url)
The temporal min of the WTA outputs yields the dendrite’s spiking output. This reduces the dendrite’s output to a single spike indicating that the input pattern was a match for at least one of the input segments, and the min function’s output spike indicates the strength of the match. Although the WTA followed by a min may appear redundant, determining a local WTA winner is important for learning via STDP to be described below.

**Example:** There are three state vectors in the associative memory (Figure 7) that have the A as the head: \[x = 5 \ 5 A\], \[β = 3 4 A\], and \[β = 5 0 A\]. All three can be stored in a dendrite having three segments.

**The Role of Similarity Clustering**

Note that a single dendrite performs a function similar to a “column” in the author’s prior work [35]. In the prior work, it is shown that a column (now a dendrite) is capable of online unsupervised clustering. A sequence of input patterns (spike volleys) is applied to a dendrite’s distal inputs, and the dendrite organizes the patterns into clusters based on similarity, one cluster per segment. Cluster centroids are encoded in the synaptic weights.

Clustering partitions input patterns according to similarity. This is important for systems where similar inputs should yield the same output or action. In effect, similarity clustering is a form of lossy compression. With that in mind, observe that the research benchmark environments consist of randomly selected features placed at random locations. Randomness is the enemy of compression. Given all the randomness in the specification, clustering has little to work with – there are few similarities upon which clustering can be based. Consequently, in the research benchmark, every cluster has a single member, so a segment holds a single state vector.

In the general case, however, segments hold cluster centroids. For completeness general clustering is described in the following paragraphs.

Because a proximal spike is necessary and sufficient for dendritic spiking, it is the distal input patterns that are clustered, one cluster per segment. The following discussion focuses on distal spiking behavior, under the assumption that the proximal input receives a spike.

Via an unsupervised training process described below, a dendrite partitions a set of distal input patterns \(P = \{a^1, a^2, \ldots, a^p\}\) into clusters of similar patterns: \(C_i \subseteq P \; \forall \; i, j: i \neq j \; C_i \cap C_j = \emptyset\). As commonly defined, the centroid \(c^k\) of cluster \(C_k\) is an element-wise average of all the members of \(C_k\).

With \(p\) distal inputs and \(q\) segments, online unsupervised clustering within a dendrite is performed by passing input volley \(a\) through a \(p \times q\) synaptic crossbar. In the example of Figure 17, a \(4 \times 3\) crossbar is used. The crosspoints hold weights \(0 \leq w_{ij} \leq w_{\text{max}}\). The weights are established via an unsupervised learning process (STDP) that adjusts the weights so that each segment is eventually associated with a cluster i.e., there is a strong correlation between a segment’s synaptic weights and a cluster’s centroid. (The learned “centroids” are only close approximations and are not strictly-defined mathematical centroids.) For a given input \(a\), dendrite segments using ramp-no-leak response functions evaluate centroid distances, and the segment associated with the nearest centroid fires first. Subsequently, WTA inhibition selects the nearest centroid, say \(c^k\), and a spike on output \(z_k\) serves as a cluster identifier.

Furthermore, the spike time on output \(z_k\) indicates the relative distance from the nearest centroid. Hence, the inference process exhibits “radial basis function behavior” [31] by determining 1) cluster membership, indicated by the presence of a spike, and 2) the distance from the cluster’s centroid, indicated by the spike’s relative time.

Most real-life situations are not random, and the non-randomness leads to clustering opportunities. Follow-on research will focus on more realistic problems based on non-random environments where the full capabilities of clustering can be better exploited.
**Excitatory Neurons**

A complete excitatory neuron is composed of one to many dendrites, combined by a temporal min function (Figure 18). Each of the dendrites has a different proximal input. Consequently, the time of an output spike at z indicates the strength of the best match from among all the segments.

![Excitatory Neuron Diagram](image)

**Figure 18.** An excitatory neuron is composed of multiple dendrites combined via a temporal min function. a) drawn as a tree, b) drawn in a more compact fashion.

### 6.2 Spike Time Dependent Plasticity (STDP)

State vectors are stored in the associative memory (PL) via a learning process. As noted above, they are organized into segments according to the proximal input head. Storing state vectors in PL is not a one-shot process as in the abstract associative memory described above. Rather, they are stored via a learning process where multiple applications (learning episodes) may be required before the state vector is solidly entrenched in a segment.

Because a proximal input spike is necessary and sufficient for dendritic activity, it was observed that all the proximal synapses that receive any input spikes quickly converge at $w_{\text{max}}$. Hence, to expedite simulation, they are initialized at $w_{\text{max}}$ and they follow the response function for $w_{\text{max}}$. 
The STDP update functions for distal synapses are given in Table 2. Weights saturate at 0 and at $w_{max} = 8$. Recall, however, that the response functions are scaled by the number of distal input bundles (4 in the research benchmark).

**Table 2. STDP update function for distal synapses.**

| input conditions | weight update |
|------------------|--------------|
| $a_i \neq null$  | $a_i \leq z_j$ | $\Delta w_{ij} = +2$ |
| $a_i \neq null$  | $a_i > z_j$   | $\Delta w_{ij} = -2$ |
| $a_i = null$     | $z_j = null$  | $\Delta w_{ij} = 0$  |
| $a_i = null$     | $z_j \neq null$ | $\Delta w_{ij} = -2$ |
| $a_i = null$     | $z_j = null$  | $\Delta w_{ij} = 0$  |

For distal synapse training, initial weights prior to training are $w = w_{max}/2$. Consider the following simple training sequence.

First, when a never-before-seen input volley is applied to an initialized dendrite and the proximal input receives a spike, all segments not yet associated with a volley will yield the same potential. Because there is a tie, it is up to 1-WTA inhibition to select one. As stated earlier, the tie breaker first selects spikes with the highest body potential at the time the spike is generated, and any remaining ties are broken systematically by selecting the neuron with the lowest index (in the simulator).

In this particular case, it will go to the second tie breaker. After the tie is broken, the distal synapses receiving input spikes have their weights incremented by 2 (up to $w_{max}$). All synapses not receiving a spike have their weights decremented by 2 (down to 0). After this is done, the given input pattern has been captured by the winning neuron, and synapses matching the input volley have weight $w = w_{max}/2 + 2 = 6$. All other distal synapses connected to the same segment have their weight reduced $w = w_{max}/2 - 2 = 2$. This means that the selected segment is biased toward its state vector and away from any other state vector.

### 6.3 Minicolumns

Thus far, the way state vectors are stored (learned) in PL has been described, as has the way inference is performed to determine the best match. What remains is producing the PL output(s) for the best match. This process occurs at the minicolumn level.

A minicolumn consists of a number of parallel neurons feeding tWTA inhibition (Figure 19). The minicolumn in the figure generates the displacement $\Delta x$ for one of the head values. For implementation simplicity and ease of understanding, we would like the input $\Delta x$ and the $i_{\Delta x}$ output to use the same encoding. This can be guaranteed by forcing $\Delta x_i$ to be the only distal input that can be stored in a segment for $i_{\Delta x_i}$, and it is for that reason only $\Delta x_i$ synapses are present in the $i_{\Delta x_i}$ segments, as shown by the dark grey shaded synapses.
Figure 19. Example minicolumn composed of single dendrite neurons. The synapses for the envId are not shown.

For the research benchmark, the complete PL (Figure 20) contains three minicolumns. One for each of the three state vector elements that are accessed during orientation (envId) and navigation (Δx and Δy). This design uses multiple-dendrite neurons that have a dendrite for each of the proximal inputs. An alternative equivalent design, along the lines of one used in [22], uses single-dendrite neurons and many more minicolumns.
Figure 20. Full PL implementation for research benchmark application. In this example, multi-
dendrite neurons are organized as three minicolumns.

7. envId Loop

The envId loop implementation is similar to the tail loop implementation. The only wrinkle is that the
envId loop performs environment disambiguation during orientation, so the loop may contain unions of
environments rather than a single environment. This means the envId code is not necessarily 1-hot. It is
k-hot if the actual environment has been narrowed down to k possibilities.

Reducing multiple envIds to a single envId is a small scale version of disambiguation performed in [22].
This is illustrated in the example of Figure 8. In that example, PL is initially queried to determine
i_envId. If there are multiple entries with the same number of matches with the key, then iWTA produces
multiple spikes. These enter the envId loop by the same process used in the tail loop. As the agent moves
on to new features, all the components of the query key are filled so a complete key is available for
matching during a query. This means that only environments with exactly the same edges in their graphs
pass through a narrowing process. Unless the graphs are identical, this process will eventually result in a
single unambiguous envId.
8. Research Benchmark Simulations

Benchmark simulations primarily demonstrate that the proposed macrocolumn implementation works as described. Generating a variety of results for large ranges of parameters and configurations aren’t especially enlightening because of the artificial nature of random environments and because a naive agent is used solely for the purpose of testing the query mechanism for supporting navigation, not for achieving some specific objective.

8.1 Benchmark Details

For a set of benchmark simulations, 40 different environments are laid out on a 30×30 grid. Each environment contains the same 10 features randomly placed. The exploration path is designed to visit all 10 features four different times in a random sequence.

Simulation consists of an exploration phase and an orientation/navigation phase. Simulations are further divided into episodes and steps. An episode is an exploration or navigation of a single environment and consists of a sequence of 100 steps, or gamma cycles. It is capable of moving an arbitrary number of grid cells in a single step (gamma cycle). Because there are 40 environments, the simulation performs 40 exploration episodes followed by 40 navigation episodes.

After orientation, the agent initiates queries and movements by first querying with a random target to find an \( i_\Delta x \) and \( i_\Delta y \). If there is a match in the PL, then the agent performs the inferred movements. Otherwise, it tries another random target. Because the graph is connected, it will eventually succeed.

Because there is no similarity clustering and each segment is expected to hold a single state vector, the number of segments per dendrite is an important parameter. To get a handle on the number of required segments, the pseudo-random exploration paths for all 40 environments were first generated so that the eventual environment graphs could be determined. To account for random variations the maximum number of segments over all three minicolumns was determined (skipping the details), and the number of segments in the simulator is set to that upper bound.

8.2 Simulation Results

Because the number of segments allocated is sufficient to fit all the state vectors (by design), the main purpose of simulation is to demonstrate that the model functions as expected. A measured statistic is the number of steps it takes for orientation to complete; see Figure 21. Because there is significant variation in the environments and movements depend on random number generator seeds, three different seeds were simulated. Overall, most episodes infer the correct environment/feature in a few steps, although there are some outliers that take as long as 16 steps. Once oriented, accesses to PL during simulated navigation are always correct.
If the number of segments in the design is not sufficient to hold all the state vectors, i.e. the segments are over-subscribed, then navigation failures may occur. A failure is detected, for example, if the macrocolumn gives displacements to a specific feature, but when the agent moves by the indicated displacements, the expected feature is not present. However, the macrocolumn is designed to handle such failures in a graceful way: when an error is detected the macrocolumn re-sets itself to the null state, re-orient itselves, and continues on.

The plot in Figure 22 is for macrocolumns containing fewer segments than the calculated necessary number for all the state vectors. The statistic is the fraction of steps after the initial orientation where the agent is correctly oriented: its state vector indicates the correct envId and the feature for its current cell location is correct. For these simulations, the calculated lower bound on the sufficient number of segments per dendrite is 16. Figure 22 gives results for segment counts from 16 down to 4. Even with only 4 segments per dendrite, the agent is correctly oriented over 92% of the time.
9. Concluding Remarks and Future Research

The basic macrocolumn learns and constructs directed graphs that describe 2-d spatial environments. The macrocolumn maintains a state vector that provides all the necessary information for learning a graph edge. It then uses the graph to navigate as part of the thought process.

The model is based on spiking neurons that perform temporal coding and processing. There are two main places temporal processing takes place. One is in autaptic loops which provide a non-synaptic way of maintaining state. A temporal mechanism provides a way of entering new values into the loop. The other is in the RIF neurons where the time of the output spike indicates the strength of an input match. In this work, temporal coding does not pass beyond the gamma cycle boundary. Rather, normalization to binarized signals is done at the gamma boundary before volleys are fed back to macrocolumn inputs. Furthermore, all of the macrocolumn inputs are binarized. This is a step back from more aggressive temporal coding that crosses the gamma boundary as done in the author’s prior work [36].

What this paper demonstrates is a bare-bones macrocolumn that supports the basic functions of exploration, orientation, and navigation. If the proposed macrocolumn accurately captures the fundamental way the neocortex works, then it can serve as a foundation for ever more elaborate brain-like cognitive systems. The bare-bones macrocolumn can also be characterized by the things it does not do:

1) no similarity clustering
2) no multi-macrocolumn regions or hierarchies
3) it supports only a 2-d grid that mirrors a 2-d environment
4) no invariance for scaling, orientation, distortions

Another limitation of the research reported here is that the agent is naive and is not constructed of model neurons. Following subsections discuss future research targeted at the above limitations.

9.1 Agents

In the research benchmark, the agent performs a minimal function: it moves about randomly from feature to feature. This is adequate for demonstrating the macrocolumn’s ability to provide navigational information (displacements), but not much else. Furthermore, in simulation the agent is a piece of Matlab code.

A topic for near-term research is the development of agents constructed of model neurons that strive to achieve specified objectives within a given environment. As a simple example, the presence of cheese at a grid point can be one of the learned features. Then, if the agent’s objective is to find food and the mouse is placed at an arbitrary grid point in a previously-learned environment, it moves around sensing the various features and then, having gotten its bearings, head along a near-optimum path (a sequence of edges in the environment graph) directly to the cheese. Similarly, if one of the features is water and another is a warm place to sleep, an agent can direct movement along a previously learned path. Here, the important enhancement to the model agent is the ability to learn near-optimal paths by using reinforcement learning (RL). The agent uses RL to learn the best next target for determining the next feature along a path to find food, water, or a warm grid point.

An additional enhancement is implementing an agent using minicolumn-like spiking neural networks that learn via a biologically plausible STDP version of RL. An RL agent capable of balancing an inverted pendulum is described in [8]. That implementation uses simple SRM0 neurons and does not distinguish distal and proximal synapses. By adding proximal synapses, the same RL agent learns multiple objectives as shown in Figure 23.
Figure 23. An agent that uses reinforcement learning to follow optimal or near-optimal paths in order to achieve an objective.

An RL agent as just described is still relatively simple. What does an agent look like that drives language or mathematical processing based on macrocolumn frames?

Say the metaphorical space aliens come to earth and examine a state-of-the-art CMOS microprocessor. Most of what they will see is SRAM – multi-level on-chip caches and predictors. And they may observe that the more SRAM, the better the performance (sometimes by huge amounts depending on working set sizes). They would then endeavor to discover how an SRAM works, motivated by the belief that it is the most important part of the computer. It is an essential part, to be sure, but one can argue that the CPU is where the real magic takes place – it uses SRAM as a large data structure to support its operation.

The point is that even if we discover a complete and accurate model for the way the neocortex works, we may have only discovered the way “SRAM” works. It may be that the agent is primarily “CPU”. And following this line of reasoning, the real key to understanding cognitive thought may reside in the agent, not the neocortex.

At a minimum this suggests that reverse-architecting research should incorporate the co-design of neocortical architecture and agent architecture.

9.2 Similarity Clustering

The macrocolumn developed in this document allocates a segment to each distal input pattern (state vector). Although the macrocolumn segments perform similarity clustering, strictly speaking, there is only one pattern per cluster. In a realistic application, the segment is trained to match multiple patterns forming a cluster, not just a single pattern. The synaptic weights in a segment define the cluster centroid. Consequently, all the similar patterns belonging to the same segment strongly match the segment’s weights and cause a dendritic spike. Segments perform the same function as the SRM0 neurons in [35][36], and implementing similarity clustering in dendritic segments should be straightforward.

Furthermore, for more realistic clustering benchmarks, environments with structured, non-random characteristics should be modeled. In other words, realistic problems where similarity clustering will offer significant benefits.

9.3 Regions

In this paper a single macrocolumn is considered. The next layer of abstraction (Figure 1) is a region formed by interconnecting multiple macrocolumns. Figure 24 is a simple schematic for such a method.
This type of structure is considered by Lewis et al. [22]. For larger problem sizes, multiple macrocolumns operate in parallel, with each working on part of the larger problem – a simple example is a very large environment where macrocolumns track overlapping sections of the environment and where they collectively reach agreement regarding the overall environment identity via a form of distributed consensus (“voting” [22]).

Within a macrocolumn, the consensus mechanism uses a second set of distal inputs. Excitatory neurons have two biologically distinct sets of distal inputs: basal and apical. The distal inputs discussed in this paper are basal; distal inputs used for distributed consensus are apical [22].

**Figure 24.** A region consists of multiple macrocolumns (MC) that observe and move through overlapping receptive fields. A voting, or consensus, network implements distributed consensus to determine a region-wide object identifier.

### 9.4 Hierarchies

Hierarchies are more easily discussed in terms of “objects” rather than “environments”. For example, identifying an object by moving it around in one’s hand is a function very similar to navigating an external environment. Objects can be composed of features that are sub-objects having spatial relationships [17]. When reaching into a fishing pack for a fly box, the sharp corners, hinges, and clasp are identifiable objects by themselves based on their shapes. A macrocolumn (or macrocolumns) associates each of these objects with an object Id. At the next higher level of the hierarchy, the fly box as a whole is composed of these objects (now sub-objects) that have spatial relationships with each other, and at the higher level, the fly box has its own object Id. In Figure 11, the envId is a macrocolumn output that can be used for constructing such hierarchies.
9.5 **Invariance**

To achieve the brain-like ability to recognize objects, symbols, etc. there must be some method for implementing invariance with respect to scales, rotations, and distortions. (It is already invariant with respect to spatial shifts). It seems that a good place to provide invariance is in the GD/path integration part of the macrocolumn. This suggests that various types of conformal mappings are built into the GD/path integration system. Going one step further, if the grid cell interconnection structure is wired-in, then that leaves path integration as the prime candidate for implementing invariance.

9.6 **Non-sensorimotor Grids**

Sensorimotor actions involve sensing and moving through a physical environment as studied here. For this kind of activity, modeling grid cells and path integration is relatively straightforward because both the external environment and the grid are 2-d (or 3-d) coordinate systems.

However, “all thinking is movement” extends well beyond a 2-d or 3-d physical systems. For constructing macrocolumns suitable for higher level, more abstract cognitive tasks, the research becomes much more challenging. For such higher level cognitive tasks such as language processing, both a grid and a method for path integration will need to be discovered. Solving problems of this type and implementing them with macrocolumn architecture will require significant research breakthroughs.
10. References/Bibliography

[1] Abeles, Moshe. “Synfire chains.” Scholarpedia 4, no. 7 (2009): 1441.
[2] Bi, Guo-qiang, and Mu-ming Poo. “Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type.” The Journal of neuroscience 18, no. 24 (1998): 10464-10472.
[3] Bichler, Olivier, et al. “Extraction of temporally correlated features from dynamic vision sensors with spike-timing-dependent plasticity.” Neural Networks 32 (2012): 339-348.
[4] Bohle, Sander M., Han La Poutré, and Joost N. Kok. “Unsupervised clustering with spiking neurons by sparse temporal coding and multilayer RBF networks,” IEEE Transactions on Neural Networks, 13, no. 2 (2002): 426-435.
[5] Butts, Daniel A., et al. “Temporal precision in the neural code and the timescales of natural vision.” Nature 449, no. 7158 (2007): 92-95.
[6] Fries, Pascal, Danko Nikolić, and Wolf Singer. "The gamma cycle." Trends in neurosciences 30, no. 7 (2007): 709-716.
[7] Gardner, R.J., Hermansen, et al. Toroidal topology of population activity in grid cells. Nature 602, 123–128 (2022).
[8] Gerstner, Wulfram, and J. Leo Van Hemmen. “How to describe neuronal activity: spikes, rates, or assemblies?” In Advances in neural information processing systems, (1993): 463-470.
[9] Gerstner, Wulfram, et al. “A neuronal learning rule for sub-millisecond temporal coding.” Nature 383, no. 6595 (1996): 76-78.
[10] Gerstner, Wulfram, et al. "Eligibility traces and plasticity on behavioral time scales: experimental support of neohebbian three-factor learning rules." Frontiers in neural circuits 12 (2018): 53.
[11] Gray, Charles M., et al. “Oscillatory responses in cat visual cortex exhibit inter-columnar synchronization which reflects global stimulus properties.” Nature 338, no. 6213 (1989): 334.
[12] Güttig, Robert, and Haim Sompolinsky. “The tempotron: a neuron that learns spike timing–based decisions.” Nature neuroscience 9, no. 3 (2006): 420-428.
[13] Guyonneau, Rudy, Rufin Vanrullen, and Simon J. Thorpe. “Neurons tune to the earliest spikes through STDP.” Neural Computation 17, no. 4 (2005): 859-879.
[14] Hawkins, Jeff, and Sandra Blakeslee. On intelligence. Macmillan, 2004.
[15] Hawkins, Jeff, and Subutai Ahmad. "Why neurons have thousands of synapses, a theory of sequence memory in neocortex." Frontiers in neural circuits (2016): 23.
[16] Hawkins, Jeff, Subutai Ahmad, and Yuwei Cui. "A theory of how columns in the neocortex enable learning the structure of the world." Frontiers in neural circuits 11 (2017): 81.
[17] Hawkins, Jeff, et al. "A framework for intelligence and cortical function based on grid cells in the neocortex." Frontiers in neural circuits 12 (2019): 121.
[18] Hawkins, Jeff, A Thousand Brains: A New Theory of Intelligence, Basic Books, 2021.
[19] Hodgkin, Alan L., and Andrew F. Huxley, “A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve,” The Journal of physiology 117.4 (1952): 500.
[20] Hopfield, J. J. “Pattern recognition computation using action potential timing for stimulus representation.” Nature 376 (1995): 33.
[21] Kheradpisheh, Saeed Reza, et al. “STDP-based spiking deep neural networks for object recognition.” Neural Networks 99 (2018): 56-67.
[22] Lewis, Marcus, et al. "Locations in the neocortex: a theory of sensorimotor object recognition using cortical grid cells." Frontiers in neural circuits 13 (2019): 22.
[23] Mainen, Zachary F., and Terrence J. Sejnowski. “Reliability of spike timing in neocortical neurons.” *Science* 268, no. 5216 (1995): 1503-1506.

[24] Maass, Wolfgang, “Networks of spiking neurons: the third generation of neural network models.” *Neural networks* 10.9 (1997): 1659-1671.

[25] Masquelier, Timothée, and Simon J. Thorpe. “Unsupervised learning of visual features through spike timing dependent plasticity.” *PLoS Comput Biol* 3, no. 2 (2007): e31.

[26] McCulloch, Warren S., and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity." *The bulletin of mathematical biophysics* 5, no. 4 (1943): 115-133.

[27] Morrison, Abigail, Markus Diesmann, and Wulfram Gerstner. "Phenomenological models of synaptic plasticity based on spike timing." *Biological cybernetics* 98, no. 6 (2008): 459-478.

[28] Moser, E. I., E. Kropff, and M. B. Moser. "Place cells, grid cells, and the brain's spatial representation system." *Annual Review of Neuroscience* 31 (2008): 69-89.

[29] Mountcastle, Vernon B. “The columnar organization of the neocortex.” *Brain* 120, no. 4 (1997): 701-722.

[30] Mozafari, Milad, et al. "Bio-inspired digit recognition using spike-timing-dependent plasticity (stdp) and reward-modulated stdp in deep convolutional networks." *Pattern Recognition* 94 (2019): 87-95.

[31] Natschläger, Thomas, and Berthold Ruf. “Spatial and temporal pattern analysis via spiking neurons.” *Network: Computation in Neural Systems* 9, no. 3 (1998): 319-332.

[32] Saada, Ravit, et al. "Autaptic excitation elicits persistent activity and a plateau potential in a neuron of known behavioral function." *Current Biology* 19, no. 6 (2009): 479-484.

[33] Sanders, Honi, et al. "Grid cells and place cells: an integrated view of their navigational and memory function." *Trends in neurosciences* 38, no. 12 (2015): 763-775.

[34] Smith, James E. "(Newtonian) Space-Time Algebra." *arXiv preprint arXiv:2001.04242* (2019).

[35] Smith, James E. "A Neuromorphic Paradigm for Online Unsupervised Clustering." *arXiv preprint arXiv:2005.04170* (2020).

[36] Smith, James E. "A Temporal Neural Network Architecture for Online Learning." *arXiv preprint arXiv:2011.13844v2* (2020).

[37] Smith, James E. "Implementing Online Reinforcement Learning with Temporal Neural Networks." *arXiv preprint arXiv:2204.05437* (2022).

[38] Thorpe, Simon J. “Spike arrival times: A highly efficient coding scheme for neural networks.” *Parallel processing in neural systems* (1990): 91-94.

[39] Thorpe, Simon J., and Michel Imbert. “Biological constraints on connectionist modelling.” *Connectionism in perspective* (1989): 63-92.

[40] Tuckwell, Henry C. “Synaptic transmission in a model for stochastic neural activity.” *Journal of theoretical biology* 77.1 (1979): 65-81.

[41] Van Rullen, Rufin, Jacques Gautrais, Arnaud Delorme, and Simon Thorpe. “Face processing using one spike per neurone.” *BioSystems* 48 (1998): 229-239.

[42] Zador, Anthony M. "A critique of pure learning and what artificial neural networks can learn from animal brains." *Nature communications* 10, no. 1 (2019): 1-7.
Appendix 1: Grids and Path Integration

GD Neuron Implementation

The GD cells and path integration can be implemented via spiking neurons with active dendrites (Figure 25). The network shown learns to implement permutations via RL. In this case, the permutations are shifts of -1, 0, +1 in each of the two dimensions. Additional shift amounts can be implemented by increasing the number of dendrites per neuron.

Figure 25. Implementation of 2-d path integration with three active dendrite neurons.